Static Analysis of Programs with Application to Malicious Code Detection

By

Raymond Waiman Lo

B.S. (University of Hong Kong) 1986
M.S. (University of California, Davis) 1991

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Karl N. Levitt, Co-chair

Ronald A. Olsson, Co-chair

Myla M. Archer

Committee in Charge

1992

Copyright© by

Raymond Waiman Lo

1992

Raymond Waiman Lo
September 1992
Computer Science

Static Analysis of Programs with Application to Malicious Code Detection

Abstract

The goal of this research was to develop a method of detecting malicious code (e.g., computer viruses, worms, Trojan-horses, and time/logic bombs) and security-related vulnerabilities in system programs. The Malicious Code Filter (MCF) is a programmable static analysis tool developed for this purpose. It allows system administrators and security analysts to examine a program before installation, thereby avoiding any damage a malicious program might inflict. This dissertation has addressed several issues in malicious code detection:

- We investigated and classified malicious code. Based on this analysis, we developed the *tell-tale sign* approach to distinguish malicious code from benign programs. The novel idea about tell-tale signs is that they allow us to determine whether or not a program is malicious without requiring a programmer to provide a formal specification.
- We generalized program slicing to reason about tell-tale malicious properties. Program slicing produces a *bona-fide program*—a subset of the original program behaving exactly the same with respect to the realization of a specified property. By combining the tell-tale sign approach with program slicing, we can examine a small subset of a large program to conclude whether or not the program is malicious.
- We demonstrated the capabilities of the tell-tale sign approach and program slicing to detect some common UNIX vulnerabilities.
- We determined how our approach can be defeated and developed countermeasures—the *well-behavedness check*. Static analysis produces inaccurate slices on a program that has pointer overflows, out-of-bounds array accesses, or self-modifying code. Well-behavedness check applies flow analysis (integer-range analysis) and verification techniques (loop invariant generation, verification condition generation, and theorem proving) to identify such problematic cases.

The MCF is designed in a modular way so that it can serve as a platform for further malicious code research including dynamic analysis and testing-based detection techniques.

Acknowledgments

My desire to research in Malicious Code Detection is sparked by my advisors Prof. Karl Levitt and Prof. Ronald Olsson. My understanding of Malicious Code Detection, and of many issues in Computer Security and Program Verification, was greatly aided by them. They foster my idea in this dissertation, guide me through the work, point out my oversights, and give me constant encouragement. Without their encouragement and guidance, this work would not be possible. I must also thank them for spending many nights and weekends reading drafts of this dissertation.

Prof. Myla Archer is the third member of my dissertation committee. Her keen insight in the theorem proving area and her careful reading of the final draft have been invaluable.

The research reported here was performed while I was supported by the UC Regents Fellowship. My tenure as a graduate student was also supported in part by Deloitte & Touche Inc., the state of California MICRO program, Lawrence Livermore National Laboratory, and the Department of Energy.

Thanks to my fellow students at UC Davis, for thousands of interesting hours—especially Wilson Ho, Jeff Becker, Lucas Hui, Tony Leung, Sara Kavala, Chris Wee, and Rick Crawford. I am particularly indebted to Wilson who not only helps me with technical problems, but also shows me how to survive in graduate school.

Thanks to Douglas Mansur for his supports and constructive suggestions to my work.

Thanks to Susan Izumi who pointed out many errors in earlier drafts.

I must give thanks to my dear wife, Amy, for her constant love and endless support throughout these years. She never fails to lift my spirits during those dark periods.

Finally, I must give thanks to my parents. Without their support, I would never start this work.

Abstract

The goal of this research was to develop a method of detecting malicious code (e.g., computer viruses, worms, Trojan-horses, and time/logic bombs) and security-related vulnerabilities in system programs. The Malicious Code Filter (MCF) is a programmable static analysis tool developed for this purpose. It allows system administrators and security analysts to examine a program before installation, thereby avoiding any damage a malicious program might inflict. This dissertation has addressed several issues in malicious code detection:

- We investigated and classified malicious code. Based on this analysis, we developed the *tell-tale sign* approach to distinguish malicious code from benign programs. The novel idea about tell-tale signs is that they allow us to determine whether or not a program is malicious without requiring a programmer to provide a formal specification.
- We generalized program slicing to reason about tell-tale malicious properties. Program slicing produces a *bona-fide program*—a subset of the original program behaving exactly the same with respect to the realization of a specified property. By combining the tell-tale sign approach with program slicing, we can examine a small subset of a large program to conclude whether or not the program is malicious.
- We demonstrated the capabilities of the tell-tale sign approach and program slicing to detect some common UNIX vulnerabilities.
- We determined how our approach can be defeated and developed countermeasures—the *well-behavedness check*. Static analysis produces inaccurate slices on a program that has pointer overflows, out-of-bounds array accesses, or self-modifying code. Well-behavedness check applies flow analysis (integer-range analysis) and verification techniques (loop invariant generation, verification condition generation, and theorem proving) to identify such problematic cases.

The MCF is designed in a modular way so that it can serve as a platform for further malicious code research including dynamic analysis and testing-based detection techniques.

Table of Contents

List of Figures

CHAPTER 1

Introduction

1. _1

1.1. Motivation

Despite heavy dependence on computers in industry and the military, computers are notoriously vulnerable to attacks. For many years, computer scientists have foreseen the vulnerabilities of computers and recognized the danger. Recently, many attacks on computers have been observed. [Denning,attack] Recognizing the significance of such incidents, researchers and developers have built defenses against similar future attacks. The nature of attacks can, for convenience, be divided into two classes: attacks from intruders and attacks by malicious programs.

Intruders are typically experienced programmers who try to gain unauthorized access to computing resources and/or information. Some well-known cases include:

- In 1984, Ken Thompson in his Turing Award lecture [Thompson,reflections] demonstrated a way to compromise the security of a UNIX system by modifying the C compiler and introducing a trapdoor in the login program. To prevent the removal of the login trapdoor, the malicious C compiler introduces the trapdoor again when the login program is re-compiled. The malicious C compiler cannot be eliminated by re-compilation of the C compiler; it outputs itself instead of a benign C compiler.
- In August of 1986, an intruder broke into supposedly well-secured computers in the Defense Department, looking for military information. [Stoll,hacker] The saga is detailed in the book “Cuckoo’s Egg”. [Stoll,cuckoo]
- In September of 1986, an intruder systematically broke into a large number of computer systems (at least 9 universities, 15 companies, and 9 ARPANET sites) in the San Francisco Bay area. The intruder left behind re-compiled login programs to simplify his re-entrance. [Reid,reflections]

Malicious programs are programs that cause loss of confidentiality and integrity, or cause denial of resources. Common classes of malicious programs include computer viruses, [Cohen,viruses,experiment] computer worms, [Shoch,worm] and Trojan horses.

Further classes include programs that exploit security holes, covert channels, and administrative flaws to achieve malicious purposes. Some well-known cases include:

- Since 1987, a type of malicious program called a computer virus [Cohen,1984] has appeared often in the PC community. Viruses hide copies of themselves in PCs, propagate to other PCs, and, if malicious, eventually damage the software or information stored in those computers. Literally hundreds of different viruses exist on thousands of personal computers currently in use.
- In November of 1988, a graduate student accidentally released a worm program into the Internet. [Spafford,1989] [Denning,1989] [Rochlis,1989] A worm [Shoch,worm] program replicates and distributes itself to other systems on the Internet. The Internet worm infected numerous computers on the network, resulting in significant recovery costs.
- In Fall of 1989, a simple Trojan horse called “Turkey” was disguised as a free utility available from USENET news (a public bulletin board). When the program was retrieved from the network and then installed, it deleted files from the installer’s directory.

Corresponding to the nature of the attacks, research falls into two areas: intrusion detection and malicious code detection. Intrusion detection [ides,lunt] [dids] protects a system from human intruders based on the identification of abnormal user behavior, such as a novice attempting to masquerade as a superuser, browsing, looking for interesting information in tagged files, and deviation from past behavior. Intrusion detection mechanisms do not need to identify all improper behavior; [IDW,workshop,1990] a tool with 95% detection rate per session identifies nearly all intruders because a human intruder eventually exhibits some detectable behavior if he keeps on using the system. Malicious code detection protects a system from malicious programs based on the identification of abnormal behavior of programs. Malicious code detection is more demanding than intrusion detection; a 95% detection rate conversely means that 5% of the malicious programs go undetected into the computer system.

The art of intrusion detection is more established than that of malicious code detection--intrusion detection systems are installed, working, and have detected numerous break-ins. Comparatively, the techniques of detecting malicious code are not well developed. Without effective techniques to identify malicious programs, people have resorted to lesser methods such as emergency policies, a policy prohibiting importation of code, security clearance for employees, and standardized malicious code databases. Emergency policies diminish the impact of successful attacks. A policy prohibiting code importation reduces the chance of accidently admitting malicious code. Security clearances reduce the chance that malicious code will be written by an employee. Standardized malicious code databases facilitate the distribution of known malicious code signatures and the detection of known malicious code. Although these methods serve well for deterring malicious code entry and for diminishing the impacts of successful attacks, they do nothing to solve the basic problem--the identification and elimination of malicious code from computer systems.

Virus scanners are the only automated tool available nowadays for malicious code detection. The principle is to scan binary programs against a database of machine code sequences that are found in known viruses. When a new virus is found, a programmer must examine the virus and determine a sequence to be included into the database. Therefore, the database of a virus scanner must be updated frequently. Even so, virus scanners fail to detect certain viruses that use stealth techniques and polymorphic viruses that have no fixed code sequence. The idea of scanning is not very useful for detecting general malicious code because identical time-bombs, worms, or Trojan horses are unlikely to be found at different sites.

This dissertation addresses the malicious code problem by developing a more general and reliable technique to detect and identify such code. The plausibility of malicious code detection has been controversial. Some researchers have proved that the problem of malicious code detection is undecidable, while other researchers think that malicious code detection is practical, believing that the class of malicious code not detectable by sufficiently sophisticated methods is insignificant in the real world. We agree that dedicated hackers can produce malicious code defiant of any particular malicious code checker. What we can hope to develop is a tool that makes the defiance extremely difficult. Pro-detection researchers have proposed many different methods for malicious code detection. Usually these methods can detect a limited class of malicious code. We propose a more general detection method using tell-tale signs based on static analysis. We also identify ways (dubbed “stealth malicious code”) to defeat our techniques and develop more advanced techniques (the well-behavedness property) to overcome stealth malicious code. We present an overview of our approaches and techniques in Chapter 2.

1.2. Our Approach to Detect Malicious Code

We use a filter tool, Malicious Code Filter (MCF), to discriminate malicious code from normal programs. MCF uses static analysis to evaluate code for tell-tale signs. The *tell-tale* *signs* are properties used for exposing malicious code. For example, a tell-tale sign for discerning viruses is that the virus code may have the appearance of being patched into the infected program. MCF, by analyzing data dependence between modules, can find isolated program sections indicating potential virus code. Another tell-tale sign is that a virus propagates by modifying executable programs. Determining the files or programs that a program could modify identifies the presence of a virus.

Program verification verifies the entire program with respect to its specification. MCF does not attempt to determine the meaning of whole programs, just the properties that can lead to the discovery of malicious code. Following Suzuki’s terminology, [Suzuki,array] where he says that array bound checking is a weak correctness property, tell-tale properties are *weak* properties. The significance of weak properties is that we are no longer required to work with the whole program and the complete specification; instead we concentrate on the program portion relevant to the weak properties. Using *program* *slicing*, we produce a bona-fide program--a subset of the original program behaving exactly the same with respect to the computation of a specified weak property. These program slices are significantly smaller and simpler than the originals. This method facilitates the analysis of large and complicated programs.

However, program slicing does have some disadvantages. Program slicing is based on data flow analysis; [Kennedy,data,flow,survey] therefore, its accuracy depends solely on the accuracy of data flow analysis. Traditional data flow analysis expects the analyzed program to satisfy certain conditions. Devious programmers may write programs that do not satisfy these conditions to fool the data flow analyzer and, indirectly, our filter tool. We call these programs *stealth*[1] programs. We have identified some stealth techniques. We have also defined the *well-behavedness* property. We believe that a well-behaved program cannot use stealth techniques to fool a malicious code detector.

The task of identifying malicious code is conveniently divided into three subtasks:

- slicing of the program with respect to the tell-tale signs
- validation of well-behavedness
- verification of the program slice with respect to the tell-tale signs

The architecture of our MCF tool, shown in Fig. 1.1, has modules for translating to intermediate form and for analyzing data flow. The other modules perform the three subtasks mentioned above: program slicer, well-behavedness checker, and symbolic evaluator.

---

[1] We borrow the term *stealth* from stealth PC viruses that hide their presence from virus scanner. Stealth malicious code prevents its discovery by using various techniques to hide from a malicious code detector.

- **Translator.** The translator transforms a source program into an intermediate form and performs some simple analyses such as generating the program and the call graphs. Some syntax analysis is performed, such as type checking, checking for peculiar conversion of pointers, etc. Errors[2] in translation are reported to the human security analyst. Because all other parts of the tool work on an intermediate form, MCF can be extended to other programming languages by building another translator.
- **Flow Analyzer.** The flow analyzer takes an intermediate form program and performs various flow analyses such as alias analysis, reaching definitions, definition-use chains, and integer-range analysis. The flow analysis framework [Kildall,unified] implements the analyses. The result of the analysis is stored back with the intermediate form program (augmented program graph). The flow information will then be used by the program slicer and well-behavedness checker.
- **Program Slicer.** The program slicer isolates the relevant portion of the program called the *program* *slice*. Program slicing is useful during the debugging, [Agrawal,debugging] testing, [Weiser,debugging] and maintenance portions of the software life-cycle. [Horwitz,integrating] The flow analysis framework [Kildall,unified] [Kam,framework] helps to compute the alias information and the data dependence information. Given a property expressed in terms of statements and program variables, we look for the statements affecting the value of the property directly (through data flow) or indirectly (through control flow). The resulting program slice, which can be used to determine the given property of the original program, is generally much smaller than the original.
- **Well-Behavedness Checker.** The well-behavedness checker shows that the program slice actually represents all computation of the weak property (to protect against stealth programs). The correctness of the program slice depends largely on the correctness of the data flow information. To ensure the well-behavedness, we perform an exhaustive alias analysis to make sure that no array overflows and all pointer de-references refer to some data object. Reference of code as data, potential array overflow, and de-reference of pointers with invalid addresses are reported to the security analyst.
- **Symbolic Evaluator.**[3] A symbolic evaluator executes a program (or a program slice) symbolically. The result is a symbolic expression that can be used to determine whether certain tell-tale properties are satisfied. This step is carried out with varying degree of difficulty. For example, we can slice with respect to the “file access” property. If a program opens no files at all (unless it writes directly to disk), its program slice is empty and we can easily conclude that the program cannot be a virus.

---

[2] We detect syntax errors, illegal type conversion, and undeclared variables.

[3] The symbolic evaluator has not been built in this prototype.

1.3. Summary of Contributions

The theme of this dissertation is the discrimination, using static techniques, of malicious programs from normal programs. The contributions include the development of the approach to detect malicious code, the reasoning about programs, and the implementation of the filter tool. The implementation is novel in the sense that it integrates different static analysis techniques for security analysis.

- **Malicious Code Detection.** We identify tell-tale signs to differentiate malicious from benign programs. We apply program slicing so that we can focus the analysis on a bona-fide program that is much smaller than the original.
- **Stealth Malicious Code.** We identify possible compromises (stealth malicious code) and develop defenses against them.
- **Program Model for Reasoning.** We define a program model on which static analysis, flow analysis, and program slicing can be precisely performed. We define the well-behavedness property, which relates analyzed programs to our program model. We develop static techniques to check the well-behavedness.
- **Program Analysis Tool.** We design an intermediate form corresponding to our program model. We include a translator from the C language into the intermediate form, a flow analysis framework for alias analysis and various data flow analyses, a loop invariant generator based on template matching, and a program slicer for the intermediate form.

The prototype implementation is a static analysis tool enabling a security analyst to interactively examine programs. A program slicer in the tool helps the analyst identify program statements corresponding to a specified property. The tool uses a library of malicious code properties to simplify the detection of malicious programs. Finally, a well-behavedness checker searches for potential stealth programming techniques.

1.4. Organization

The rest of this dissertation is organized as follows:

- Chapter 2 presents the overview of malicious code detection. It includes examples demonstrating the application of the tool to detect malicious code and identify anti-detection techniques.
- Chapter 3 describes the static analysis part. It argues the importance of the global analysis and well-behavedness check to malicious code detection. It also presents the various components in the tool.
- Chapter 4 presents the program model.
- Chapter 5 describes some interesting issues in the design of the translator.

- Chapter 6 discusses the flow-analytic framework and defines various flow analyses with the framework.
- Chapter 7 covers program slicing.
- Chapter 8 presents well-behavedness checking.
- Chapter 9 presents a taxonomy of malicious code and security holes, and develops a set of tell-tale signs to detect such code and holes.
- Chapter 10 discusses the strengths and weaknesses of this approach, and the application of the techniques to other areas. It also concludes the dissertation and describes future directions.

We have tried to make each chapter independent of the others. Readers mainly interested in malicious code detection can read Chapters 2 and 9. Readers interested in an overview of the analysis tool without getting into too many details can read Chapters 2, 3, 9 and 10. Readers interested in the detailed implementation can read Chapters 4 through 7. Readers interested in applying our tool to other applications such as software reliabilities (e.g., identification of potential run-time errors) and program verification (e.g., verification of weak correctness properties) can read Chapter 8.

CHAPTER 2

Overview of Malicious Code Detection

2. _2

This chapter gives an overview of the malicious code detection problem and compares existing approaches, including run-time techniques, comparison-based techniques, dynamic analysis, and verification, to our static analysis approach.

In Section 2.1, we define malicious code and malicious code detection. In Section 2.2, we discuss the feasibility of malicious code detection. In Section 2.3, we examine the various approaches and their limitations, and then propose and develop our approach using tell-tale signs. In Section 2.4, we give an overview of the tell-tale approach: its mechanism and its effectiveness in detecting malicious code. Apart from the tell-tale signs and the filter, there are two very important points to consider. The first question, discussed in Section 2.4.1, is to define the kinds of malicious code the tell-tale approach can and cannot handle. The second question, answered in Section 2.4.2, is whether this approach can be defeated and at what cost in writing malicious code. Section 2.5 ends with several examples of malicious code detection. Details of the tell-tale approach are illustrated with an example in Section 2.6.

2.1. Malicious Code Detection

Recall from Section 1.1, *malicious* *programs* are programs intending to damage computer systems, release or modify information without authorization, or deny services. *Malicious* *code* *detection* is the identification of such programs.

Malicious code may be defined by its exact implementation (sequence of code). Virus scanners detect known viruses by scanning for such sequences, but they suffer from a lack of generality in being usable to detect new malicious code.

Malicious code may be defined by its behavior and detected by identifying such behavior. However, defining malicious behavior, in general, is futile because maliciousness is a context-dependent concept. For example, the act of erasing a hard disk can either be malicious or benign, depending on the situation. If a user intends to erase the hard disk using the disk formatter, the act is benign. If a computer virus initiates the hard disk erasure, the act is malicious. How can we detect malicious code when we are not able to tell which behavior is malicious! We compromise and err on the safe side. We identify some normal programs as malicious (false positive), but do not classify any malicious programs as normal (false negative). With our approach, the disk formatter is classified as malicious and then carefully scrutinized. We call those identified programs *suspicious* and the detection *suspicious* *code* *detection*. In subsequent discussions, we do not distinguish malicious and suspicious, since suspicious code is what we can handle; therefore we use the two words interchangeably.

2.2. Feasibility of Malicious Code Detection

Cohen [Cohen,theory] proved that malicious code detection is undecidable. However, this undecidability does not mean total unfeasibility. Crocker and Pozzo [Crocker,filter] claimed that malicious code detection in limited cases (using specification and program verification) is possible. We agree with Crocker and Pozzo and further elaborate on the feasibility of malicious code detection with our approach.

Cohen [Cohen,theory] showed that distinguishing viruses from normal programs is unsolvable. His proof is very similar to the proof of the halting problem. The undecidability of the halting problem states that no Turing machine, hence no computer program, can tell whether a program, executed on certain input will stop or run forever. Similarly, Cohen concludes that determining whether a program does or does not contain a virus by examining its code is undecidable. Cohen's proof, outlined in Fig. 2.1, only exposes the ineffectiveness of this particular approach to malicious code detection, but he does not show that malicious code detection in general is undecidable when maliciousness is defined differently, thus permitting a different approach (i.e., algorithm) for malicious code detection.

Assume the existence of a decision procedure $D(Prog)$ that returns $true$ if $Prog$ is a virus, $false$ otherwise. Then we construct a virus CV, known as the contradictory-virus, as follows:

```
program CV :=
{ ...
        main-program := {
                if not D(CV) then {
                        infect-executable;
                        if trigger-pulled then do-damage;
                }
                goto uninfected-program;
```

```
            }
    }
```

Proof:
We apply D on program CV. $D(CV)$ either return $true$ or $false$.

Case 1: $D(CV) roman
By definition of CV, infect-executable is executed in CV. Hence CV executes as a virus and contradicts $D(CV) = false$.

Case 2: $D(CV) roman
By definition, infect-executable and do-damage are not executed. Hence CV never shows any viral activities and contradicts $D(CV) = true$.

Hence virus detection is undecidable.

Cohen has implicitly defined a program to be a virus when it can *actually* infect other programs and initiate damages. Take the contradictory virus program CV shown in Fig. 2.1 as an example. Suppose D(CV) is never false. Then CV is not a virus as defined by Cohen, even though it contains viral code (infect-executable and do-damage). We believe it is unreasonable to classify a program containing reachable or unreachable viral code to be benign.

Although Cohen shows that it is impossible to discern a virus using his definition, we can always identify a larger program class, called *suspicious* programs, using a more relaxed definition, i.e., the existence of viral code. Thus, CV is easily identified as a suspicious program.

Formal *specification* and *verification* provided an alternative in distinguishing benign programs from suspicious programs, as proposed by Crocker and Pozzo [Crocker89] in 1989. They define a virus as a program that changes other programs. The program developer identifies the *specification* as a restriction to specify what a program is permitted to change. A program is accepted if it makes changes only according to the specification. *Verification* demonstrates that the program actually behaves according to the specification. Their proposal employs a *filter* program, which verifies the program in question with respect to its specification.

2.3. Common Detection Techniques

2.3.1. Run-time Approaches

There are many ways to detect malicious code. The most common approaches involve running the program to see whether it shows any viral activities. Although run-time approaches are easy to implement and do detect malicious activities, they have several drawbacks. First, it is impossible to detect malicious code without exposing a system to potential damage using run-time approaches. Second, run-time techniques only detect and then inhibit malicious programs' activities. They cannot identify the presence of malicious code when the code is dormant. For example, a program working fine for a long time can suddenly behave maliciously. Third, when a run-time tool identifies a problem, it either stops the malicious program or asks for human attention. Of course, it cannot ask the user at every file access because there are too many accesses. Fourth, for systems running unattended, run-time approaches are simply not viable.

- Hardware techniques [Joseph88] [Mahmood88] use auxiliary hardware to compare the execution of a program to a predetermined control flow graph, and stop the program when the control flows do not match. Because of the granularity of monitoring and the latency in identifying discrepancies, damage could have occurred before the program is stopped.
- Load-time checking techniques [Cohen,checksum] [Cohen,models] [Pozzo,encryption] do not allow a program to execute if it fails to pass certain tests--typically cryptographic checksums. However, they require a *clean* copy of the program to generate the checksum. Moreover, they generate an unacceptable amount of overhead if software decryption is used.
- Operating system modification techniques query the users at run-time for all file modifications, [Cohen,defenses] or require users to identify the programs that can write to their files. [Karger87] [Wichers90] These methods require a lot of user intervention and most of the users' effort is wasted if no virus is present.

2.3.2. Static Analysis

In contrast to run-time approaches are the static approaches. Static approaches perform the analysis without executing the program. Therefore, they do not have the problems associated with run-time approaches. However, static analysis is harder to implement. Current static methods are comparison-based. They fall into the following three general categories according to what the program is compared to:

- with a *clean* copy of the program. It guarantees the program is not changed as compared with the clean copy. Still, this method does not show that the clean copy is, itself, free of malicious code.
- with known malicious code (e.g., virus scanning). It locates previously known viruses, but cannot detect slightly mutated viruses. Therefore, it is vulnerable to intentional attacks: an experienced programmer can easily modify an old virus to evade detection.
- with a formal specification. A program is compared (verified) against its specification. Unfortunately, both the formal specification and verification of programs are at best difficult. Furthermore, commonly used programs often have no specification, and are very unlikely to be verified.

2.3.3. Dynamic Analysis

Dynamic analysis [crawford,lo,synthesis] combines the concepts of testing and debugging to detect malicious activities by running a program. A program is typically executed in a clean-room environment while being monitored by a programmable debugger [Dalek,programmable,debugger] for suspicious behavior (tell-tale signs). The analysis is in general more reliable than in ordinary run-time approaches because data are generated systematically to test the program. [Hamlet] Test coverage analysis will also reveal parts of programs not covered by the testings. Compared with static analysis, dynamic analysis is less reliable because testing can never be exhaustive.

2.3.4. Code Screening by Human

In the previous three sections, we have discussed automated malicious code detection. The most well-known examples of malicious code analysis are, in fact, performed by experienced hackers by hand, e.g., the analysis of the Internet worm, [Rochlis,microscope] the identification of known viruses and analysis of new viruses. [Fischer,virus,center,research]

Although a human can reason about a program in detail, he is weak in examining code and data that are spatially or temporally separated, and also has difficulty in handling large amount of information at one time.[1] A malicious program may exploit the human weaknesses by obfuscated programming techniques such as using macros, overflowing pointers, writing self-modifying programs, or installing sections of malicious

---

[1] Typical software engineering methodologies aim to reduce the spatial or temporal separations. Object-oriented approaches group data declarations and the functions operating on the data together. Divide-and-conquer reduces the amount of information a programmer needed to handle at one time. Debugging tools tend to assist in the understanding of the temporal aspects of a program.

code in spatially separated parts of the program.

A human can be fooled by his knowledge. When he looks at a procedure, he assumes that it always returns. When he looks at a procedure with a familiar name, e.g., qsort, he may assume that the procedure performs a sorting, even though qsort is malicious. Therefore, a malicious code may use familiar variable names and procedure names to camouflage itself.

Finally, humans err. Thus the result is not reliable.

2.3.5. The Ideal Approach

To sum up, an automated malicious code detection tool, which is better than existing tools and approaches, needs to be developed. In addition to detecting a large class of malicious code, some important criteria for a useful detector are:

(1) It performs static analysis to detect dormant malicious code that might not be exposed by running the program.

(2) It detects both known and unknown malicious code of present classes. It is not comparison-based.

(3) It does not require a *clean copy* or a specification of the program.

(4) It is practical (in terms of time and effort) to use; therefore, it should not use formal specification and complete verification.

(5) It is applied with minimal human intervention; therefore, it should be automated.

(6) It errs on the safe side (false positive) at worst.

(7) It is programmable to deal with new classes of malicious code.

2.3.6. Our Approach

Our approach closely approximates the ideal approach. It is difficult, however, to simultaneously satisfy criteria (2) and (5). First, a fully automatic tool requires the ability to foresee all types of malicious code. Defining malicious code is still a research problem; it is very difficult to know what kinds of malicious code will be written in the future. Therefore, the tool must rely on the security analyst for the definition and description of new malicious code. We allow the analyst to define tell-tale signs discerning these new malicious code so that our tool can detect them. Second, even after defining the malicious code properties, it is still very hard to automate the identification of these properties or produce a tool that understands a program and identifies the malicious properties. So we adopt a semi-automatic approach--program slicing and symbolic execution. Program slicing reduces the size of code to be understood. Symbolic execution, under the direction of the analyst, learns how the program executes.

2.4. Detection Using Tell-Tale Signs

In everyday life we can distinguish, without specification, coffee from water by their colors. Most of the time we can tell a cat from a dog by its appearance. Analogously, some program properties allow us to distinguish malicious programs from benign programs. We call these properties the *tell-tale* signs. The idea is to program a *filter* to identify the tell-tale signs without the need to give a specification of the program being analyzed. Nevertheless, we may mistakenly identify some normal programs as malicious. The goal is to minimize such mistakes.

We can use tell-tale signs to identify computer viruses. A typical virus behavior is shown in Fig. 2.2. It first examines some writable and executable files. If the file has enough empty space at the end of the program, the virus infects the program. The infection is started by copying the viral code, which is written in a relocatable manner, to the end of the text segment; registering the original entry point so that control is passed back to the original program when the virus finishes executing; and modifying the entry point of the victim program to viral code. A program infected by this virus has the following tell-tale signs.

- The program has duplicated system calls, e.g., access, open, read, lseek, write, close. A virus is usually self-contained and cannot rely on the system calls in the infected program. [Snitch]
- The program has “strange” data dependencies and isolated/independent code. No shared (global) variables and parameters are passed between the viral procedure and other program procedures.
- The program accesses its own text segment as data. When the virus copies itself to other programs, it reads the viral code from its own text segment. Reading the text segment is a rare activity in normal programs.
- The virus has strange file accesses. The virus opens and writes to executable files, normally only done by linkers.

The above discussion relates to general virus detection methods. PC viruses, in particular, may have other tell-tale signs. A program running on a PC can easily bypass the operating system and access the disk directly. This direct disk access is one tell-tale sign. Other types of malicious code such as Trojan horses and time-bombs have their own unique tell-tale signs. Details will be given in Chapter 9. Furthermore, we have to design and implement a programmable filter to discern malicious programs with the various tell-tale signs. The modules of the filter are discussed in Chapters 4 through 8.

2.4.1. Effectiveness of Tell-tale Detection

We first divide malicious code into two classes: *generic* and *specific*. We then divide the programs being analyzed into two other classes: *programmable* and *non-programmable*. This provides us with four combinations.

N-G: Discrimination of Non-Programmable programs and Generic malicious code. N-S: Discrimination of Non-Programmable programs and Specific malicious code. P-G: Discrimination of Programmable programs and Generic malicious code. P-S: Discrimination of Programmable programs and Specific malicious code.

When a program's execution (function) is determined by its inputs, we call it *programmable*, and when its execution is not, *non-programmable*. Programmable programs usually perform different functions for different inputs; the inputs instruct the programs what to do. Non-programmable programs perform similar functions with different inputs; the inputs are just data to be processed. Language interpreters (Icon, Lisp, AWK), command interpreters (Bourne Shell, C-Shell), and some configurable programs (Emacs, Sendmail) belong to the programmable class. Other programs, such as many UNIX commands (cat, ls, who, finger), whose function is consistent with different inputs, belong to the non-programmable class. The major distinction is that with programmable programs, you can ask them to do whatever you want by supplying the appropriate input.

Generic malicious code can be attached to any program without severely affecting functionality of the host, e.g., a trapdoor added to the login program, a virus attached to the editor program, or a time-bomb embedded with the spreadsheet program. Generally, an infected host performs its original function except for a minor loss of performance due to the malicious code. The opposite of generic is *application* *specific*; this class of malicious code only attaches itself to particular programs. Application-specific malicious code attacks by modifying the program's functional behavior. For example, an application-specific malicious code can change a plane's flight control program enough to crash the plane. Flight control programs can be attacked in many ways, such as invoking “retract the landing gear” during landing and “drop the nose” at takeoff. Perhaps the malicious code is only invoked at a particular airport. Real-time control programs may be compromised with the insertion of a simple delay loop. Navigation and guidance systems are vulnerable to inaccuracy in the trigonometric functions.

The reason we make these distinctions is that the effectiveness of the malicious code detection tool varies with these different cases. We divide them into four groups, as shown in Fig. 2.3. We can specify the *proper* behavior of non-programmable programs. We can identify generic malicious code because of the way its behaviors is expressed in the program. We can predetermine the tell-tale signs covering different species of malicious code, e.g., the modification of executables (virus), or the bypassing of authentication (trapdoor). However, we cannot determine whether or not programmable programs are malicious. Their actual behaviors depend not only on the programs, but also on their inputs. The situation is best illustrated with the Bourne Shell and a shell-script. The Bourne Shell interprets and carries out the operations of the shell-script, which can be either malicious or benign. We cannot generate tell-tale signs for a application-specific malicious code without knowing exactly what the infected program is and in what ways the infected program is damaged.

The tell-tale approach can discern generic malicious code in both programmable and non-programmable programs (P-G and N-G). With the aid of the security analyst identifying the important security properties, this approach can discern specific malicious code from non-programmable programs (N-S). The discrimination of programmable programs and specific malicious code (P-S) is difficult. It requires formal specification and general program verification because a malicious program can exploit anything wrong in a programmable program to evade detection. Furthermore, demonstrating the non-maliciousness of programmable programs does not imply that no malicious action is possible; one must show that the input to P, the interpreted program, is also benign. The effectiveness of the tell-tale approach is summarized in Fig. 2.4.

2.4.2. Stealth Malicious Code

If we review the current tools that detect viruses, we see that these tools can be defeated. The reason is simple enough: if a malicious programmer knows how the detection tools work, he can devise a virus to evade detection--e.g., the PC stealth viruses. The following are some stealth techniques applicable to existing detecting tools:

- Virus scanners examine a list of virus *signatures*; each signature is basically a sequence of machine instructions. If a hacker knows the signature, he can rewrite that portion of the virus to evade detection.
- A new type of virus, polymorphic virus, can change their code whenever they propagates. Their signatures are so general that a virus scanner either fails to detect polymorphic viruses or generates too many false alarms.
- The 386-cache virus takes advantage of the cache of the Intel 80386 processors; the processor assumes the program text segment is not modified and therefore cache-able. The virus modifies itself so that the code in main memory differs from the code in cache. Consequently, the PC has two versions of the programs: a benign one in the main memory and a malicious one in the cache memory. When someone tries to examine or single-step the programs inside a debugger, the benign one is being examined. When the program is executed directly, the malicious version is executed. This stealth technique requires a self-modifying program.
- The Snitch tool [snitch] detects viruses by assuming a virus carries its own operating system calls and checking for such duplication. The malicious programmer can write a virus, with some difficulty, using the existing operating system calls.

The above stealth techniques do not work against our tool; however, the following are several possible ways to defeat our tool (and others too), and we have come up with some remedies.

Attacks	Remedies
• Use collaborating programs. The malicious activities are distributed among two or more programs.	Collaborating malicious code needs to have at least one program that commits the damage. By programming our tool to search for the code that commits the damage, we can identify one of the collaborating programs.

• Learn our tell-tale signs and avoid them.	Our defense against the avoidance technique is, first, to select some hard-to-avoid tell-tale signs, e.g., the determination of file accesses. Second, we predict the evasions and write tell-tale signs for them. For example, a virus evades detection from Snitch by linking with existing system calls. The linking involves a search for the necessary routine. We can detect the linking by the tell-tale sign "searching in code space". Since neither "duplication of operating system calls" nor "searching in code space" is common, the combination is a fairly effective sign for virus detection. Third, we believe if we keep playing the game of measures and counter-measures, the malicious code will get so large and so complicated that its existence will become very obvious.
• Use obfuscated programming techniques, including self-modification, overflowing arrays, executing data, tampering directly with the stack, etc. Section 3.2.2 contains a few examples.	Obfuscated programming techniques usually lead to the violation of the C language semantics. Therefore, a part of our tool is dedicated to verify the conformation to the language semantics. We call the conforming programs well-behaved. Chapter 8 gives the details.

We think that a good malicious code detection tool should disclose ways that it might be compromised because a malicious code writer will surely learn of existence of a detection tool and of its detection method. Once a method to defeat a tool is found, the method can be automated to convert existing malicious code to undetectable malicious code. For example, virus scanners are found to be useless against polymorphic viruses. A toolkit which converts existing PC viruses to polymorphic viruses has been developed and exchanged among virus writers. [Solomon,stealth] Furthermore, the detection tool should also identify cases in which its result might be unreliable, such as programs with out-of-bound array accesses. See Section 3.2.2 and Chapter 8 for more details.

2.5. Examples of Malicious Code Detection

In the following sections, we briefly present how some malicious code and vulnerabilities can be identified, and how some stealth techniques are detected. The filter tool is applied to detect some malicious code: the RUSH HOUR virus [Burger,1988] and the generic time-bomb program; and some system vulnerabilities: the UNIX #chown()# bug and the UNIX #ftp# bug. [usenet,altsecurity] Currently, the filter prototype works with simple C source programs, but does not work with MS-DOS executable or complicated system programs. The following is based on hand-simulation of the tool. More about the implementation will be discussed later.

2.5.1. The RUSH HOUR Virus

The RUSH HOUR virus program [Burger,1988] was developed and published as a demonstration program for computer viruses. It is intended to harmlessly show the danger of viruses to computer systems. The virus only lodges itself in the MS-DOS German keyboard driver KEYBGR.COM. When the virus is in the system, it searches the current directory for the keyboard driver every time the user accesses the disk. The virus, which camouflages itself as a keyboard driver, intercepts all MS-DOS system calls. The infecting action is triggered by the load-and-execute system call. After being triggered, the virus tests the KEYBGR.COM on the specified drive and infects it, if it has not already been infected.

We can use our tool to look for file access system calls in MS-DOS system files and device drivers, which should not have any.

2.5.2. The Generic Time Bomb

Time bombs are malicious programs whose damage is triggered at a certain time. A generic time bomb, as shown in Fig. 2.5, first reads the current time, and then compares it with the triggering conditions. If the triggering conditions are satisfied, the time bomb performs the damage. Although the triggering algorithm is rather standard, the damage varies.

```
time-bomb:
  now = gettimeofday();
  if (trigger-time(now))
    do-damage;
    ...
```

The security analyst can program the filter tool to recognize (slice) for such execution patterns. The execution of a time-dependent statement is rather suspicious because most utilities uses the current time only for auditing. There are just a few exceptions, e.g., the UNIX 'make' program, incremental backup procedures, or editors like emacs. The detection procedure is summarized as follows:

(1) Identify the #gettimeofday()# [BSD,manual] system call.

(2) Perform a forward data-flow slicing starting from the #gettimeofday()# system call for the computations using the result of that system call.

(3) Identify the set of statements conditionally executed upon the time-dependent values by slicing.

(4) Interpret the sliced program. If the output is large (which is suspicious enough), the analyst may perform slicing again on some specific damage-producing code.

2.5.3. UNIX Change Ownership Bug

The UNIX file system has two important attributes for access control: the owner-id identifying the file owner, and the group-id used for delegating file accesses to a group of people. The access control for each file has three levels: the permissions for its owner, the permissions for the users who are in the group specified by the file group-id, and the permissions for all others.

The file owner or the superuser can change the group-id and all the access permissions of a file, and all others cannot. In the 4.3 BSD system, the #fchown()# system call has a bug allowing one user to change the group of a file owned by another user. If the file is world readable and group writable, a user can gain the write-privilege by changing the group of the file.

The system call #fchown()# takes a file descriptor as a parameter and changes the ownership or group of the file associated with the file descriptor. #fchown()# calls #chown1()# to effect the change. #fchown()# does not test for the access rights of the file being changed, and #chown1()# does not test the access rights correctly. Figure 2.6 shows the sliced fragment of #chown1()# that tests the access permissions.

The source code related to change of file ownership is large. We use slicing to identify the access validation code.

```
/* Extracted from chown1() in
```

```
*   ufs_syscalls.c   7.3 (Berkeley)  4/18/87
*
*   ip->i_gid is the group ownership of the file.
*   gid is the second parameter in the chown() system call.
*   groupmember(gid) returns true if gid belongs to the group the process.
*   suser() returns true if the process is owned by the superuser.
*    u.u_uid is the owner of the process.
*
*   The bug can be corrected by adding the check:
*       if (u.u_uid != ip->i_uid && !suser()) return (u.u_error);
*/
  if     (gid     !=     ip->i_gid     &&     !groupmember((gid_t)gid)     &&     !suser())
                                                                  return (u.u_error);
```

2.5.4. ftp Security Bug

The ftp program associated with Sun UNIX 3.2 has a bug that allows a user, say A, to switch to any other user provided that A has successfully logged into the system once. The essential stripped code of the ftp validation, written in YACC, is shown in Fig. 2.7.

```
cmd:                                              USER SP username CRLF
                                                          {  set new user id
                                          the variable logged_in is not resetted }

                                                 | PASS SP password CRLF
                                                           { if password is correct
                                                                                  set
logged_in                                  to                                      1
                                                                                else
                                                                                  print
error                                                                        message
                                                                                   }

                                             | STOR check_login SP pathname
                                                               { call check_login
                                                               if it returns 1 then
                                                                                  write
file pathname to remote host  }

check_login:                                                                   { if
(logged_in)
                                                                           $$ = 1;
                                                                      else
                                                                           $$ = 0;
                                                                          }
```

The scenario is to log in with a valid userid and password; ftp sets the #logged_in# variable to 1. Then the hacker logs on with a *victim* userid and a wrong password; ftp prints an error message, but does not reset #logged_in#. When the user wants to store a file with *victim* privilege on the system, the ftp runs #check_login# to validate the privilege. #check_login# checks #logged_in# and returns 1, so ftp assumes the access is valid and performs the unauthorized store.

This bug can be identified with static techniques in three ways. The first one is to check for a data-flow anomaly in the data dependence graph. From this, we see that the variable #logged_in# is never initialized in the program. The second method is to expand #logged_in# symbolically to see how it is computed. The symbolic output will affirm that its value is set with a correct password, but nothing is done with an incorrect password. The third method is to examine the authentication routine for an internal states, i.e., dependence on the previous iteration. Password authentication should not have state variables other than the userid.[2]

2.6. Using the Filter Tool

To illustrate how to use the filter tool, let us examine the W-virus[3] shown in Fig. 2.8. The W-virus is a simple UNIX virus, with important compilation information and magic constants (see program comments in Fig. 2.8) removed to make reconstruction extremely difficult. The salient features of a virus, e.g. replication, are still there. Some of the easier to implement tell-tale signs for the virus are:

(1) The virus uses the asm statement, asm("mov a6@(4), a6@(-XXX)"), to embed an assembler instruction at line 73 of the C program. Asm statements are rarely used in typical C programming.

(2) The program accesses its own text segment. The access is through pointer #j# at line 82. Pointer #j# is set up at line 77. When the virus copies itself to other programs, it reads the viral code from its own text segment. Again, normal programs rarely access their own text segment as data. Using pointer mapping analysis, we detect that #j# does not point to any variables, but it is de-referenced in the program.

---

[2] The general idea is to detect authentication without providing a password.

[3] W-virus is an experimental UNIX virus developed for research purposes.

(3) The virus has strange file accesses at line 36 and 37. The virus opens and examines all files in the current directory.

The first tell-tale sign will be identified at the time of translation to intermediate form. The second sign will be identified when the tool verifies the well-behavedness of the program. The third sign is determined using slicing with respect to file open.

```
001   /*  This is a Sun-3 Unix Virus for demonstration purpose.
002       It only infects executables in the current directory.
003       Four essential parts of the virus has been removed from this source
004       to make reconstruction of this virus very difficult.  */
005
006   #define PAGESIZE 8192              /* Sun 3 running OS 3.4 */
007
008   main (argc, argv)
009   int argc;
010   char *argv[];
011   {
012       struct direct dir;
013       int dirfd;
014       int offset;
015       char *p = (char *) &offset;
016
017       /* Cannot use string in this program, because the string address
018          generated by the compiler will not be position independent.  All
019          data used by this program must be stored in the stack. */
020       p[0] = ’.’; p[1] = 0;
021
022       /* Check the current directory for executable file */
023       dirfd = open (p, O_RDONLY, 0);
024       offset = 0;
025       for (;;) {
026           register int i;
027
028           lseek (dirfd, offset, 0);
029           i = read (dirfd, &dir, sizeof(dir));
030           if (i == 0)
031               break;
032
033           /* For each file in the directory, check if it is writable and
034              executable.  All other files are ignored. */
035
036           if (access (dir.d_name, W_OK | X_OK) == 0) {
037               register int fd = open (dir.d_name, O_RDWR, 0);
038               struct exec header;
039
040               if (fd >= 0 && read (fd, &header, sizeof(header))) {
041
```

```
042     /* Only interested in demand-paged executable, because they
043        have trailing gaps in the text segment. */
044     if (header.a_magic == ZMAGIC) {
045         char buf[PAGESIZE];
046         register int i;
047
048         /* load the last page of the text segment, and see if
049            there is a large enough empty space. */
050         lseek (fd, N_TXTOFF(header)+header.a_text-PAGESIZE, 0);
051         read (fd, buf, PAGESIZE);
052         for (i=PAGESIZE-1; i>=0; i--)
053             if (buf[i] != 0)
054                 break;
055
056         /* we need only 2 Kbytes */
057         if (i <= (PAGESIZE - 2048)) {
058             char *j;
059             int entry;
060             int old_entry;
061             char mesg[20];
062
063             /* I need to get the starting address of this
064                virus, which is different on each victim.
065                Therefore this address must be calculated on the
066                fly. */
067             /* get return address to j. The magic number XXX
068                is obtained by two-pass compiling, i.e., compile
069                the program to assembly language and then find
070                out the offset from the frame pointer (a6)
071                corresponding to "j".  Adding new variables will
072                change this value. */
073             asm ("movl a6@(4), a6@(-XXX)");
074
075             /* 4 bytes before the return address is the current
076                address for main */
077             j = (char *) *((int *) (j - 4));
078
079             /* Copy myself to the victim's text segment's last
080                2 Kbytes. */
081             for (i=(PAGESIZE - 2048); i<PAGESIZE; i++, j++)
082                 buf[i] = *j;
083             lseek (fd, N_TXTOFF(header)+header.a_text-PAGESIZE, 0);
084             write (fd, buf, PAGESIZE);
085
086             /* save the location of "main" of the victim. The
087                magic number ZZZ is the offset from "start" where
088                the address of "main" is stored. */
089             /*
090                 Lines 89 through 91 are removed to
091                                 prevent reconstruction of the virus. */
092
093             /* replace this address by the entry pointer of the
```

```
                       virus */
                    lseek (fd, header.a_entry + ZZZ - N_TXTADDR(header) +
                          N_TXTOFF(header), 0);
                    entry = N_TXTADDR(header) + header.a_text - 2048;
                    write (fd, &entry, 4);

                    /* The "rts" instructions of this virus will be
                       hand-replaced by a "jmp" instruction.  The
                       target of this jmp is the victim’s main program.
                       We need to write the entry point of this "main"
                       as the operand of the "jmp", so that control
                       will be passed back to "main" after the virus
                       has done its job.
                       The number YYY is the offset from the beginning
                       of the virus routine where the victim’s entry
                       point should be written.  It is obtained by
                       two-pass compilation. */
                    lseek (fd, N_TXTOFF(header) + header.a_text - 2048
                          + YYY, 0);
                    write (fd, &old_entry, 4);

                    /* Tell the poor guy which program has been infected.*/
                    write (2, dir.d_name, dir.d_namlen);
                    mesg[0] = ’ ’; mesg[1] = ’h’; mesg[2] = ’a’;
                    mesg[3] = ’s’; mesg[4] = ’ ’; mesg[5] = ’b’;
                    mesg[6] = mesg[7] = ’e’; mesg[8] = ’n’;
                    mesg[9] = ’ ’; mesg[10] = ’i’; mesg[11] = ’n’;
                    mesg[12] = ’f’; mesg[13] = ’e’; mesg[14] = ’c’;
                    mesg[15] = ’t’; mesg[16] = ’e’; mesg[17] = ’d’;
                    mesg[18] = ’.’; mesg[19] = ’0;
                    write (2, mesg, 20);
                }
            }
        }
        close (fd);
    }
    offset += dir.d_reclen;
}
close (dirfd);
```

The tool has many steps to execute. The preparation stage involves:

(1) translating the program into intermediate form; some of the misuses of type conversions and pointer computations are identified; control flow dependence explicitly stated; information of variables (such as pointer, array bounds, local versus global variable) explicitly stated; complex statements reduced to simpler statements.

(2) performing alias analysis. The traditional naming of variables--involving scope rules and structure members--is not convenient for alias analysis. Data is now referenced in units called *objects*. An object can be a local variable, an array, a structure, or the command line arguments passed to the main procedure. Alias analysis computes the objects that a pointer may reference, thus producing the correct data dependence.

(3) performing data-flow analysis. The goal is to identify the data dependence and control dependence (see Section 4.3.4) for the program. Suppose a statement X references variables whose values are defined at statement Y. Then X depends on Y.

(4) slicing the program. Based upon the data flow analysis, the definitions used and their corresponding statements are collected.

If coming from source, the translation to intermediate form would expose the use of asm statements, [4] raising the security analyst's attention. If coming from object, it is quite impossible to analyze embedded assembly statically because the assembly is extremely compiler dependent.

The alias analysis step reveals any pointers pointing to the code segment or containing an invalid address when the pointer value is not an object. The anomalous access to code segment is identified.

With the use of slicing, the tool identifies the program text corresponding to the property "files accessed". The program slice, as shown in Fig. 2.9, consists of 8 lines out of the 132 line virus in Fig. 2.8. This slice is an example of slicing being done on the intermediate form rather than the source program. Further analysis, either static or dynamic, is done on this program slice. The conclusion is that all files in the current directory may be modified, indicating very suspicious activities. Slicing is an important technique in reducing the work needed to examine a program.

We can try to slice the program with respect to the time-bomb properties, first slicing with respect to the #gettimeofday()# system call. The slice is the empty program. Therefore, this program does not access time at all. We can also try to slice with respect to the "change of protection states", i.e., the #chown()# and #chmod()# system calls, to see whether they are used. Since the program uses neither of them, the program slice is empty with respect to the "change of protection states" property.

---

[4] Our translator works on preprocessed C programs; therefore, the asm statements cannot be hidden using macro statements.

```
020        p[0] = ’.’; p[1] = 0; 023        dirfd = open (p, O_RDONLY, 0); 024        offset =
0; 025        for (;;) { 028            lseek (dirfd, offset, 0); 029            i = read (dirfd, &dir,
sizeof(dir)); 037                register int fd = open (dir.d_name, O_RDWR, 0); 129        }
```

The next stage is the verification of well-behavedness, preventing the tool from being blessing a malicious program. This is even harder than the first stage, but it is important because the validity of the first stage depends largely on this one. Nevertheless, we achieve a mostly automatic proof that no array or pointer overflow exists. Some unproven cases[5] are highlighted for further analysis.

- Array overflow is handled by range analysis; generating the lower and upper bounds of integer expressions. The range of the subscript expression is compared against the bounds of arrays.
- Pointer overflow is handled similarly. Pointers to simple objects or members of objects have no variable arithmetic operations; thus, they are trivially handled. We represent a pointer to an array with the alias part and the offset part. The alias part specifies the array a pointer may reference; the offset functions as the subscript part. The offset part is processed with range analysis in a manner analogous to that used to detect array overflows.

The process for validating array and pointer accesses involves:

(1) mapping pointer references into array and subscript expressions.

(2) generating the verification conditions for array accesses.

(3) generating simple loop invariants using templates.

(4) using flow analysis to propagate and refine integer ranges.

(5) validating correct array accesses.

(6) transforming remaining accesses into the Boyer-Moore theorem prover[6] format and attempting to prove them with the theorem prover.

(7) outputting accesses not verified.

---

[5] The generated loop invariant is not strong enough or the values of the subscript expression cannot be sufficiently described using the propositional logic and linear inequalities.

[6] We only use the propositional and linear arithmetic part of the theorem prover.

The security analyst must examine those accesses not verified in the last item. It could simply be that our tool cannot generate a strong enough assertion or the complexity of the generated assertion is beyond the automatic capability of the Boyer-Moore theorem prover. It could be a program bug, which may or may not be exploited for security violation. It could be a malicious program, altering the data flow or control flow of the program to evade detection. Chapters 3 and 8 provide more details.

CHAPTER 3

Overview of Static Analysis

3. _3

This chapter gives an overview of our static analysis tool MCF. We assume that readers are familiar with basic compiler techniques. [Ullman,compiler] In Section 3.1, we introduce static analysis techniques. In Section 3.2, we identify some differences between traditional static analysis and security-related static analysis, and introduce the concept of well-behavedness. In Section 3.3, we introduce our static analysis tools--the translator, the flow analyzer, the program slicer, and the well-behavedness checker; the four components work close together. We also explain the relationship between these four components. In later chapters, we further explain the details of these components individually.

3.1. Background

Static analysis involves techniques used for examining programs without executing them--the analysis is on the program text. Static analysis includes syntax analysis, flow analysis, [Kennedy,survey,flow] symbolic execution, [Select] and even program verification. [Branstad] Syntax analysis (e.g., the UNIX lint [1] utility) examines the parse tree of a program to identify anomalies such as type mismatches and inconsistencies in function calls. Flow analysis computes properties, which cannot be derived from the program syntax alone, for compiler optimizations such as constant propagation, reaching definitions, common sub-expression elimination, and dead variable elimination. Symbolic execution often generates expressions to describe the relationship among program variables; it often finds applications in program debugging, documentation, and maintenance. Program verification tries to show that a program's specification is valid for all of its possible executions; automated verification tools tend to use other static analyses (cheaper methods) to uncover more information about the program before proving its correctness. For example, program verification depends on user-supplied loop invariants, but the discovery of

---

[1] lint performs mostly syntactic analysis, but it also performs some simple semantics analysis, such as identifying unreachable code and loops not entered at the top.

such invariants is difficult. To assist in automating loop invariant discovery, German and Wegbreit [german,inductive] use weak interpretation (flow analysis) to identify simple inductive invariants. These invariants might be too weak, but do provide a good starting point to generate stronger invariants.

Syntax analysis, flow analysis, symbolic execution, and program verification represent a spectrum of static analysis techniques with increasing computation complexity, increasing amount and precision of gathered information, and increasing level of human intervention during the analysis. In the extreme case, static analysis cannot be automated; the user directs the analysis in program verification. Automated static analysis is often a trade-off between the amount and precision of the information gathered and the tractability of the problem. Our analysis tool provides automated static analysis, and allows the analyst to direct the application of such analysis.

3.2. Static Analysis for Security Purposes

We apply static analysis techniques to determine if a program contains code that could violate a security policy. Although static analysis techniques have been used for program optimization for many years, the requirements for security analysis are different. In program optimization, the analyzer provides accurate information only when the program is well-behaved. With security analysis, the analyzer must identify the cases when the flow analysis may be affected by programs that are not well-behaved. As briefly described in Chapters 1 and 2, the well-behavedness, and therefore the correctness of the analysis, is a very important aspect in computer security. We do not demand exact or minimal program slices, but require that a program slice is equivalent to the original program with respect to the slicing criteria, and we allow the program slice to err on the safe side.

3.2.1. Global Analysis

Most compilers perform only intra-procedural analysis because of the limited time allowed to be spent by the optimizer. It is safe to make assumptions about the non-interference among procedures, e.g., that local variables are not modified by other procedures. In security analysis, the analysis must be global and inter-procedural. Malicious code writers will not conform to rules of good programming practice to make our lives easier. For example, a procedure in a malicious program may interfere with other procedures through non-legitimate means.

Therefore, our analysis must deal with these situations. We define a program model, described in Chapter 4, that allows global flow analysis. Global flow analysis identifies all data flow in the program. We also justify the assumptions made by the flow analyzer using the well-behavedness verifier, described in Chapter 6.

3.2.2. Well-behavedness

As mentioned in Section 2.4.2, malicious programs may use stealth techniques to prevent security analysts or analysis tools from detecting malicious activities. A devious programmer may use macros to confuse a programmer, or he may use array/pointer overflow to confuse the data flow analyzer, he may use array/pointer overflow to change the control flow of the program or to execute data.

- **Use macros to confuse the security analyst.** The devious programmer redefines some common and benign macros or library calls to perform malicious functions. Since the redefined procedure was benign and the security analyst is not aware that the procedure has been changed, he believes the program is not malicious. This technique does not confuse our static analyzer. Our analyzer works on the intermediate form, which is generated from the macro preprocessed C programs by the translator. The program in Fig. 3.1 seems to do nothing at first glance--by examining only the main procedure. However, the #exit()# library routine has been shadowed by a macro definition. The program will print out the string #"virus"# whenever the #exit()# procedure is invoked.

```
#define exit(x) puts("virus");_exit(x)

main()
{
   exit(0);
}
```

- **Use array/pointer overflow to modify data flow.** This technique uses array or pointer overflow to confuse the data flow analyzer. The program slicer uses on data dependencies. The devious programmer may use array/pointer overflows to modify variables. The modification is not represented in the data dependence graph. In the program in Fig. 3.2, the pointer #p# is overflowed to point the string #"siruv"#. By dereferencing #p#, we can actually change the string #"siruv"# to #"virus"#. The data dependence graph shows nothing about the string modification.

```
main()
{
   int i; char *p, c;
   p = "nothing" + 8;            /* the offset 8 is system dependent */
   c = *(p+4);  *(p+4) = *p;  *p = c;
   puts("siruv");
}
```

- **Use array/pointer overflow to modify control flow.** This techniques uses array/pointer overflows to gain access to the stack and modify the return address. The execution sequence of the program is different from what is perceived by the analyst or our analysis tool. In the program in Fig. 3.3, the procedure #unreachable()# is not supposed to execute because it is not called in #main()#. However, the main procedure modifies its return address by overflowing the array #x# and replacing the return address in the stack with the address of #unreachable()#. As a result, #unreachable()# is executed when #main()# returns.

```
unreachable() {
   puts("virus"); exit();
}

main() {
   int x[1];
   /* the offset of the return address from x, 2 * sizeof(int),
         is system dependent */
   x[2] = unreachable;
}
```

- **Execute data or self-modified code.** Both our tool and the analyst examine program statements for malicious activities. The devious programmer can hide the malicious code by embedding them in the the data storage area, and then transferring control to the data. The program in Fig. 3.4 shows one way to do this.

```
/* This program executes on a Sun 3 workstation */
data[] = {
  0x4e560000, 0xdffc0000, 0x48d7, 0x4878,
   0x6487a, 0x1c4878, 0x161ff, 0xc,
  0x4fef000c, 0x4e5e4e75, 0x48780004,
   0x4e404e75, 0x76697275, 0x730a0000, 0
};

main() {
  int (*f)();
  f = (int (*)()) data;
  (*f)();
}
```

Variants of this technique are often used in self-decrypting, self-modifying, or self-decompressing programs.

We can detect these stealth techniques by specifying and mechanizing a more rigid program model. These stealth techniques fail if the analyzed program satisfies the following requirements.

- The program does not modify its code.
- The program does not transfer control to data.
- The program does not allow modification of variables that have not been identified by the data flow analyzer.

These requirements are further translated into two properties: the well-formed and well-behavedness property. The well-formed property governs the generation of pointer values--all pointers must point to some variables or procedures, or have the null value. For example, assignments of integers to pointer variables are not allowed. The well-behavedness property states that there is no modification through overflowed arrays or pointers and no modification through procedure pointers. Therefore, all data dependence can be considered by the program slicer. If the two properties are satisfied, the program slice corresponds to the original program with respect to the slicing criteria. The function of the well-behavedness checker is to verify these properties. Details are given in Chapter 7.

3.3. Relationship of Static Analysis Components

Our tool consists of four components: the translator, the flow analyzer, the program slicer, and the well-behavedness checker. All of these components work with programs represented in an intermediate form. The translator converts preprocessed C source programs into our intermediate form. The flow analyzer implements the flow analysis framework; various flow problems can be solved. The program slicer, based on data dependence, slices the program with respect to our predefined properties. The well-behaved checker first checks that pointer references are well-formed, and then generates the verification conditions that no pointer or array overflow is possible. Our static analysis tool integrates these components together.

3.3.1. Translator

We built a translator that translates preprocessed C source programs to our intermediate form. Although the intermediate form is not a component of the static analysis tool, it is still important because the other three components of our analysis tool operate on this shared program representation. The syntax and semantics of the intermediate form is well defined. Complicated language features are simplified to ease the implementation of the static tools.

Preliminary static (syntax) analysis is performed. Information such as the procedure call graph, program flow graph, global variables, local variables, type of variables, what variables are modified in which statement, what variables are used in which statement, etc., is collected. The data flow information for the library functions is also defined to allow complete data flow analysis

For convenience of static analysis, we assign a unique identity to each computation node and a unique identity to all memory locations. For convenience of the static analyst, we keep track of the correspondence between the source program and the intermediate program.

3.3.2. Flow Analyzer

The flow analyzer computes global flow analysis for the abstracted program; the abstracted program contains the call graph, the control flow graphs (one for each procedure), and the statically assigned objects.

The flow analyzer provides the program slicer with the control and data dependence information. The flow analyzer first computes the global pointer alias information. Basically, the aliasing information allows us to transform the global data flow problem into an alias-free global data flow problem. Then the flow analyzer computes the alias-free global data flow dependence. The flow analyzer also computes the control-dependence used by the program slicer.

The well-behavedness checker generates assertions for validating array and pointer accesses. The checker uses both flow analysis (integer-range analysis) and verification techniques to validate the assertions. The flow analysis techniques require integer-range analysis, which the flow analyzer provides, to validate assertions, and the verification techniques require the computation of acyclic reaching definitions, which is also provided by the flow analyzer.

3.3.3. Program Slicer

The program slicer performs control-flow, forward data-flow, and backward data-flow slicing on the intermediate program. It needs the control-flow information, the data dependence (reaching definitions), and the control dependence (predicate regions). By providing the starting point and ending point of the execution paths, the slicer collects the nodes in the paths connecting them.

Backward data flow slicing is the most common use of program slicing. It determines the computation and variables that may affect the slicing properties. For example, slicing for the statements affecting the argument in the #open()# system call is backward data flow slicing. Forward data flow slicing identifies the statement that may be affected by the slicing property. For example, slicing for the statements affected by #gettimeofday()# system call is forward data flow slicing, as mentioned in Section 2.5.2.

The output of the slicer is a (sliced) program. The sliced output can either be a complete program or a program fragment. A complete program is compilable and executable in the traditional sense. A program fragment is a proper subset of the complete program, but it may contain some non-deterministic branch nodes where both directions can be taken. Therefore, a program fragment cannot be executed directly, but must be interpreted using a symbolic evaluator.

3.3.4. Well-Behavedness Checker

The well-behavedness checker verifies that the intermediate program conforms to our program model. It is important because our program slicing is correct only when the program is well-behaved.

The checker verifies that each dereferenced pointer expression is well-formed. A pointer expression is well-formed when either of the following conditions exist:

(1) the pointer expression evaluates to the base address of a simple variable.

(2) the pointer expression evaluates to the base address of an array or a structure and an integer offset.

When a well-formed pointer expression is dereferenced, the checker generates an assertion for the well-behavedness property. One assertion is generated for each pointer dereference. The well-behavedness property states that when the pointer expression is decomposable as in case (2), the offset must be no less than zero and less than the bound of the array or the structure.

The checker proceeds to validate these assertions. Simple loop invariants corresponding to array and pointer indexing are generated. Techniques involved are the use of T1-T2 interval analysis [Ullman,subexpressions] to identify natural loops, use of symbolic execution to identify loop induction variables and their corresponding recurrence relations, and the use of templates for the recurrence relations to generate the loop invariants.

The flow analyzer computes the integer ranges with the loop invariants and the assertions. The process finds some of the assertions valid. Finally, the checker generates verification conditions for the assertions. The verification conditions are proved automatically using the Boyer-Moore theorem prover.

CHAPTER 4

Static Model and Program Representation

4. _4

Three things are very important to our analysis tool. First is the model of computation; it defines the behavior (semantics) of the programs that we will analyze. With static analysis, it is impossible to evaluate exactly all possible execution paths of the program. We must approximate the program behavior to make the analysis tractable. We have defined a static model that enables us to reason about programs. The second important element to our analysis tool is the representation of the abstracted programs; a good representation allows effective analysis and communication among different components in our static analysis tool. Third is to verify the accuracy of the abstraction process. We reason about C source programs but abstracted using the static model; therefore the results cannot be established if the abstracted programs are not a valid implementation of the model. Programs satisfying the model are well-behaved; it is impossible to write a program that is not well-behaved in our model. The verification of the abstraction process is called the well-behavedness check. The static model, the program representation, and the well-behavedness check are closely related, therefore we will discuss them together in this chapter.

Section 4.1 defines the terms used in this chapter. Section 4.2 introduces and defines the program representation (internal form). Section 4.3 describes the static model for the program and the well-behavedness property. Finally, Section 4.4 summarizes the issues in designing the static model.

4.1. Definitions

This section introduces terms to facilitate clarity in the following sections. Although some terms appear very similar, they have slight, but important differences.

The *static* *model* defines the execution of programs for which we carry out static analysis. The model has two major differences from a dynamic model that captures the semantics of the programming language. First, it approximates the dynamic model--certain execution information is ignored. Second, extra syntactic and semantic constraints are imposed to make static analysis possible.

The *internal* *form* (representation for abstracted programs) contains a program graph, the object definitions, and computations expressed in an intermediate language. It serves two purposes. The first purpose is to provide a common data structure for representing programs among different components of the static analysis tool; C source programs are translated into the internal form before analysis. The second purpose is to define the difference between the dynamic and static models, in particular defining what information is omitted from the dynamic model.

The *intermediate* *language* is a part of the internal form. It is a Lisp-like language in which the computations are expressed.

An *abstracted* *program* is a program translated from a C source program and represented in the internal form. We use the term “abstracted program” when it is the program we want to emphasis. We use the term “internal form” when it is the form we want to discuss. Notice that an abstracted program is not necessarily a valid implementation of the static model, i.e., it does not necessarily satisfy the semantic constraints.

The *abstraction* *process* is the process of simplifying a C source program into a program in the internal form (the abstracted program). We say the abstraction process is correct when we can reason about the abstracted program using the static model. However, not all programs after abstraction are valid implementations of the static model. The abstraction process is correct when it is applied to well-behaved programs. The *translator* implements the abstraction process.

Well-behavedness, which is a semantic property, may be used to describe a C source program or an abstracted program. Well-behaved abstracted programs are valid implementations of the static model. The translator has checked the syntactic constraints using program transformation; the well-behavedness checker shows the programs actually satisfying the semantic constraints. Note that well-behavedness is defined by the semantic requirement of the static model; a different static model may need a different well-behavedness definition.

4.2. The Program Representation

We use an internal form to represent programs. From the software engineering point of view, tools developed using an internal form [Johnson,portable,theory] [Johnson,portable,tour] are more portable and more extensible. Since malicious code can be written in different programming languages, our tool must handle programs written in different programming languages and systems. Our tool has a front-end and a back-end, similar to the architecture of many multi-platform compilers.

[Tanenbaum,portable,compilers] [Ganapathi,retargetable,compiler] The front-end translates an analyzed program into our internal form; the back-end, which is more complicated, performs the analysis of the internal form. An advantage of this architecture is that we can port our tool to a new environment by only implementing a new front-end.

The internal form (program representation) is the nucleus of the system design. From the viewpoint of the front-end, the internal form should be easy to translate to. From the viewpoint of the back-end, the internal form should carry all the necessary information that could be easily retrieved by the back-end. There is no unique or best internal representation; different systems require different internal forms. It is easy to generate an abstract syntax tree or annotated parse tree [Diana] from parsers. Directed acyclic graph (DAG), in which nodes representing computations and edges representing data flow, removes irrelevant ordering information; therefore it is ideal for common subexpression elimination and instruction scheduling. A control flow graph is often used for various flow analyses. Simple tuple code (e.g. ucode, [icon,implementation] P-code, gcc's register transfer language) eases back-end code generation; the simplicity of tuples minimizes the effort in producing back-ends for different architectures. In the software engineering environment, a program dependence graph (PDG), [Ottenstein,Program,Dependence] [Horwitz,Prins,program,dependence] which carries both control and data flow information, is often used.

We have to find the internal form that best suits our goal. Our goal for the internal form is compatibility with multiple programming languages, such as C, Pascal, and 8086 machine code, in different environments. Thus, a long term goal is to build a C front-end, a Pascal front-end, and an 8086 front-end (decompiler) [Housel,decompiling] [Housel,Halstead,methodology] to translate the program being analyzed into our internal form. Our internal form should be simple and must be easily generated by the front-end. Furthermore, our back-end consists of several components and the internal form should allow communications among these components. We find that the PDG is not a suitable model because it has explicit data dependence requiring fairly complicated data flow analysis. Parse trees are not appropriate because they depend on the source language syntax. DAG is not used because of the difficulty in expressing data flow dependence across basic blocks.

Our internal form, Augmented Program Graph (APG), is a program graph in which the nodes represent computations and the edges represent control flow. APG is different from the traditional program graphs in several respects. For example, we keep the number of APG operations (opcodes) small and impose some constraints on the computation in order to simplify the static analysis. Our internal form can be "augmented" because information can be attached to the nodes in the program graph used in the static analysis. For example, the front-end generates the program graph with minimal additional information. We derive the aliasing information from the abstracted program and store the result with the nodes in the program graph. The aliasing information is used later to compute the reaching definitions. The reaching definitions are again stored with the abstracted program; they are used later for the array-bound verification.

4.2.1. Program Structure

APG is basically a program representation consisting of variables and procedures. We use a program control to describe the control flow, and a Lisp-like language to describe the computation, along with some syntactic constraints.

A program consists of a set of procedures. One of the procedures is the entry (main) procedure. A procedure consists of a set of nodes linked with edges representing flow of control, a set of local variables, a set of parameters, and an entry point. Nodes in the program are uniquely identified by the node number--node-id. Procedures are uniquely identified by the node-id of their entry nodes. Nodes contain either computations or branch statements. Each computation node modifies either zero or one variable and has exactly one outgoing edge. The outgoing edge specifies the next node to be executed. A branch node contains either a conditional or an unconditional goto statement. An unconditional goto statement does the obvious and has exactly one outgoing edge specifying the next code to be executed. A conditional branch node evaluates the branch condition and selects one of the two outgoing edges to follow.

Parameter passing is call-by-value. Return values are simple objects; returning composite objects, such as structures and arrays, is not allowed. Simple and composite objects are defined in the next section. Each procedure has one or more return statements. All return statements are explicit and have no outgoing edges.

APG has a database that stores information associated with the nodes and procedures, which are stored and retrieved by giving the node-id and the name of the information.

APG has several syntactic restrictions on the computation node, designed for more convenient flow analysis. We use program transformation to convert the abstracted program to satisfy the following constraints.

- Expressions have no side effects, but procedures can.
- At most one procedure call is allowed in each computation node. The first opcode of the node containing a procedure call must either be the procedure call operator (CALL) or the write operation (WR). (Section 5.7 discusses the program transformation in details.)
- At most one variable is assigned/modified in each computation node. However, more than one operator is allowed.

4.2.2. Data Structure

APG includes types very similar to those in C: char, int, float, pointer, array, structure, and union. The types of declared variables are maintained, but they are not used in the static model. We use the term *object* to represent the abstraction of variables and their types.

An object refers to a set of memory locations that static analysis cannot distinguish. For example, suppose P is a recursive procedure being invoked twice and X is a local variable declared in P. We use the notation P:X to refer to the two memory locations (one for each invocation of P) allocated for X collectively. P:X is an object in the static model. Another example involves arrays. The elements of an array belong to the same object because static analysis cannot determine the value of subscript expressions.

There are two kinds of objects: *simple* and *composite*. Simple objects are of the elementary types; composite objects are of constructed types built with the type operators array, structure, and union. The primary difference is that an access to a simple object always refers to the entire object. Writing to a simple object overwrites all previous information stored there; writing to a composite object may modify part of its contents.

An object may be derived from a local variable, a global variable, storage allocated from the run-time support, dynamically allocated storage, or temporary variables used in program transformation. Therefore, not all objects have corresponding variables in the source program. For convenience, each object is assigned a unique identifier, called an object-id.

To summarize the important properties of objects in APG:

- All memory locations that a program may access belong to objects.
- Allocated memory of distinct objects does not overlap.
- An object may correspond to either one or more variables, and each variable may have one or more instances.

4.2.3. Lisp-like Intermediate Language

Computations are represented in a Lisp-like language. The evaluation follows that of the Lisp language. We chose a Lisp-like syntax because the static analyzer is developed in a Common Lisp environment. We deliberately keep the number of operators in the language small--23. An interpreter can be built by defining a function for each of these operators.

Figure 4.1 has a short example of a procedure computing the factorial function. The first part of APG defines the mapping between objects and the variables. Object 0 is a global temporary introduced by program transformation. The second part of the APG is a printable representation of the APG procedure. The procedure consists of 4 statements: with node-ids from 0 to 3. The node-id is on the left side of each line. The intermediate language is on the right side. A more complete example is shown in Section 4.2.5.

```
(RET 1))(variables
   (0                                    # Mapping between variable GLOBAL name
$0)                                      and object-id.
   (1 FACT N) )

(procedure FACT (N) 0   (GOTO (> (RD (OBJ 1)) 0) 1 3) 1   (WR (OBJ 0)
(CALL FACT (- (RD (OBJ 1)) 1))) 2   (RET (* (RD (OBJ 0)) (RD (OBJ 1)))) # Procedure FACT with argument N 3
```

Figure 4.2 contains the context-free grammar for the intermediate language, defining how the computations can be constructed. Each node contains a statement. A statement can be a CALL, LIB, RET, or WR statement. The statements are expanded with other operators and expressions as described by the production rules. The boolean, arithmetic, and logical operators have their usual meanings. The OBJ operator distinguishes between integer constants and object-ids. The RD operator takes an object-id and obtains the value stored in the object; the value could be a character, an integer, a floating point number, or another object-id. The WR operator takes an object-id and a value; it stores the value into the object. The SEL operator takes two expressions. The first expression evaluates to an object-id of the structure type. The second expression evaluates to an offset within the structure. The AREF operator also takes two expressions. The first one evaluates to an object-id of the array type. The second one evaluates to an integer index. CALL and RET stand for procedure call and return, respectively. LIB is similar to CALL, but instead calls a library procedure.

```
/* Operators of the intermediate language */
%tokens RD, WR                                /* memory accessing */
%tokens +, -, *, /, %                         /* arithmetic operators */
```

```
%tokens AND, OR, XOR                          /* logical operators */
%tokens <, >, =, !=, <=, >=                   /* relational operators */
%tokens AREF                                  /* array access */
%tokens SEL                                   /* structure access */
%tokens GOTO                                  /* branches */
%tokens CALL, LIB, RET                        /* procedure calls */

/* other terminals */
%tokens PROC_NAME                             /* name of a procedure */
%tokens NODE_ID                               /* a node id */
%tokens OBJECT_ID                             /* an object id */
%tokens NUMERIC_CONSTANT                      /* a number */

statement
        :  call_statement
        |  return_statement
        |  assignment_statement
        |  goto_statement
        ;

call_statement
        :  ( CALL PROC_NAME expr_list )
        |  ( LIB  PROC_NAME expr_list )
        ;

return_statement
        :  ( RET )
        |  ( RET expr )
        ;

goto_statement
        :  ( GOTO expr NODE_ID NODE_ID )      /* conditional goto */
        |  ( GOTO NODE_ID )                   /* unconditional goto */
        ;

expr_list
        :  /* empty rule */
        |  expr
        |  expr expr_list
        ;

expr
        :  NUMERIC_CONSTANT
        |  ( OBJ OBJECT_ID )
        |  ( RD expr )
        |  ( SEL expr expr )
```

```
            | ( AREF expr expr )
            | ( unary_op expr )
            | ( bool_op expr expr )
            | ( arith_op expr expr )
            | ( logical_op expr expr)
            ;

unary_op
            :  - ;
bool_op
            :  < | > | = | != | <= | >= ;

arith_op
            :  + | - | * | / | % ;

logical_op
            :  AND | OR | XOR ;
```

4.2.4. Support for Static Analysis

An important application of the internal form is to support various static analyses of the abstracted program. Instead of defining a fixed data structure and storing the internal form in some global variables, we use access functions to obtain information about the abstracted program. The idea is similar to the abstract data type concept; the implementation of APG is hidden from the static analysis tools. Since APG is shared between components of the static analysis tool, we do not want changes in any components to affect others. In our first prototype of the data flow analyzer, we represented reaching definitions with lists. Later, we re-implemented reaching definitions using bit vectors to reduce the execution time of the data flow analyzer. Because of the use of access functions, other components of our tool are not affected by the changes in the data flow analyzer. Furthermore, APG can be easily extended because more information is presented by providing additional access functions. Conclusively, we find the access function approach beneficial.

For convenience, we divide the access function into the four groups shown in Figure 4.3.

The first group is the conversion functions, which maintain the relationship of the APG objects to C variables and APG statements to C statements. The second group is the basic functions. The information accessed through these functions is provided by the C-to-internal-form translator. The third group of functions does not relate to any particular program. It is a set of functions provided to the static analyzer in case the analyzer needs to examine the intermediate code. The fourth group is the extended functions. They provide information gathered by static analysis and stored with the APG program.

In the following sections, we will elaborate on the available access functions.

4.2.4.1. Notation

L is a line of C program.
C is a line of internal code.
N is a node-id.
P is a procedure-id (also a node-id).
T is a type or typename.
O is a object-id.
PROCNAME is the name of a procedure in the source program.
VARNAME is the name of a variable in the source program.

4.2.4.2. Conversion Functions

The relation between APG and the source program is maintained and is accessible through the following conversion functions.

(p2name P)	Returns the procedure name string.
(name2p PROCNAME)	Returns the procedure-id (the entry point of the procedure).
(o2v O)	Takes an object id as the parameter and returns the name of the variable.
(v2o PROCNAME VARNAME)	v2o is the inverse of o2v.
(n2l N)	Takes the node id as the parameters and returns the corresponding C statement.
(l2n L)	Takes a C statement and identifies the corresponding node in APG. It may find zero, one, or more than one node.

4.2.4.3. Basic Functions

Global information of the APG program is provided by the following functions.

(get-node-list)	Returns the list of node-ids for the program.
(get-proc-list)	Returns the list of procedure entry node-ids for the program. Procedures are uniquely identified by their entry node. The list is in depth-first ordering.
(get-object-list)	Returns the list of object ids for the global variables, local variables, and dynamically allocated storage.
(no-of-nodes)	Returns the number of nodes in the program.
(no-of-procs)	Returns the number of procedures in the program.
(no-of-objects)	Returns the number of objects in the program.
(code N)	Returns the Lisp-like intermediate code of the node specified by N.
(from N)	Returns the list of node-ids that precede N.
(to N)	Returns the list of node-ids that succeed N.
(inproc N)	Finds out which procedure node N is in. Returns the procedure id.
(formals P)	Returns the list of object-ids for the formal parameters of procedure P.
(locals P)	Returns the list of object-ids for the local variables declared in procedure P.
(nodes P)	Returns the list of node-ids for the nodes belonging to procedure P. The list is in depth-first ordering.
(caller P)	Returns the list of procedures which call P.
(callee P)	Returns the list of procedures called by P.
(entry P)	Gives the entry node of procedure P, which is also P.
(exits P)	Gives the list of exit nodes of procedure P, including all RET statements and the implicit exit: the end of the procedure.
(is-simple-obj O)	Returns true if O is a simple object: neither an array nor a structure.
(is-compose-obj O)	Returns true if O is a composite object: either an array or a structure.
(type O)	Returns the type or typename of the object. A typename is returned for recursively defined types. See also (expand-type).
(bounds O)	O must be of array type. It returns the size of the array.
(sizeof T)	Returns the number of bytes required for type T.

4.2.4.4. Opcode Access Functions

These functions support easy manipulation of the internal program; they are useful for defining the transfer function of flow analysis. The following functions return true when:

(is-cond-goto C)	C is a conditional goto statement.
(is-uncond-goto C)	C is an unconditional goto statement.
(is-goto C)	C is either a conditional or an unconditional goto statement.
(is-call C)	C is a procedure call; the procedure is defined in the source program.
(is-lib C)	C is a library call; the procedure is defined in the library.
(is-ret C)	C is a return statement.
(is-arith C)	The first opcode of C is one of +, -, *, /, %.
(is-rel C)	The first opcode of C is one of <, >, <=, >=, =, /= (not equal).
(is-rd C)	The first opcode of C is the memory access operator.
(is-wr C)	The first opcode of C is the write operator.
(is-aref C)	The first opcode of C is the array subscript operator.
(is-sel C)	The first opcode of C is the structure element select operator.
(is-obj C)	The first opcode of C is the OBJ operator.

4.2.4.5. Extended Functions

Using the basic functions, we can define functions to obtain various information, such as the objects used in a computation node. The information is usually computed with flow analysis. Furthermore, there is a dependency among flow analyses. For example, the reaching-definition analysis can only be computed after pointer-mapping analysis. Finally, the list of extended functions is not exhaustive. Each analysis typically adds some functionality to the language.

(object-used C)	The objects directly referenced (read) in the intermediate code C.
(lvalue E N)	The lvalue returned by the expression E; computation of lvalue requires the pointer alias table at node N.

(pmap N)	The pointer alias table at node N.
(use N)	The objects directly or indirectly used in node N.
(def N)	The objects modified in node N.
(kill N)	The definitions removed by the computation in node N.
(pdef N)	The objects that may or may not be modified by node N. For example, when a pointer is aliased to several objects. Modifications through the pointer only change one of the objects. It is equivalent to the difference of (def N) and (kill N).
(reach O N)	The definitions of object O that reach node N.
(loops P)	The loops in procedure P. Loops are uniquely identified by their entry node. They are used in identifying loop induction variables.
(lib-use LIBNAME ARGLIST N)	The objects indirectly used, through the arguments, by the library.
(lib-def LIBNAME ARGLIST N)	The objects indirectly modified, through the arguments, by the library.
(lib-kill LIBNAME ARGLIST N)	The objects, with the effects of previous definitions totally eliminated, modified by the library.
(lib-ret LIBNAME ARGLIST N)	The objects returned by the library.

4.2.5. Example

An example C program, sgrep.c, is shown in Fig. 4.4. Its APG representation, in a printable format, is shown in Fig. 4.5. Our purpose with the example is to provide an overview of APG without expecting the reader to understand how to perform static analysis with APG or how to translate C source programs into APG representations. We will further discuss the static model in Section 4.3, the translation in Chapter 5, and various analysis techniques in Chapters 6 through 8.

sgrep.c is a simple pattern-matching program. It obtains a pattern and a file name from the command line arguments. It prints the lines matching the pattern. The first part of APG is the mapping between C variables and APG objects. The rest are the APG procedures.

```
/*  sgrep

   - print out the lines in the file that match the argument.
   - bugs: does not handle lines longer than 255 characters.
*/
```

```
#include <stdio.h>
#define MAXLINELEN  256

main(argc, argv)
int argc;
char *argv[];
{
   char *pattern, *file;
   FILE *fp;
   if (argc != 3) {
      fprintf(stderr, "Usage: sgrep <pattern> <file>\n");
      exit(-1);
   }
   pattern = argv[1];
   file = argv[2];
   fp = fopen(file, "r");
   if (fp == NULL) {
      fprintf(stderr, "sgrep: cannot open file %s\n", file);
      exit(-1);
   }
   grep(pattern, fp);
   fclose(fp);
}

grep(pattern, fp)
char *pattern;
FILE *fp;
{
   char buf[MAXLINELEN];
   while (fgets(buf, MAXLINELEN, fp) != NULL) {
      if (index(pattern, buf) > 0)
         fputs(buf, stdout);
   }
}

index(pattern, line)   /* return index of s in t */
char *pattern, *line;
{
   int i, j, k;
   for (i = 0; line[i] != '\0'; i++) {
      j = i;
      k = 0;
      while (pattern[k] != '\0' && line[j] == pattern[k]) {
         j++;
         k++;
      }
      if (pattern[k] == '\0') return i;
   }
   return -1;
}
```

```
(Variables
    (0 CRT0 ARGP)
    (1 CRT0 ARGS)
    (2 MAIN ARGC)
    (3 MAIN ARGV)
    (4 MAIN PATTERN)
    (5 MAIN FILE)
    (6 MAIN FP)
    (7 INDEX PATTERN)
    (8 INDEX LINE)
    (9 INDEX I)
    (10 INDEX J)
    (11 INDEX K)
    (12 GLOBAL $0)
    (13 GLOBAL STDERR)
    (14 GLOBAL STDOUT)
    (15 GREP PATTERN)
    (16 GREP FP)
    (17 GREP BUF)
    (18 CONST "Usage: sgrep <pattern> <file>\n")
    (19 CONST "r")
    (20 CONST "sgrep: cannot open file %s\n")
    (21 POINT 6)
    (22 MAIN RET)
    (23 INDEX RET)
    (24 GREP RET)
)

(procedure MAIN (ARGC ARGV)
0   (GOTO (/= (RD (OBJ 2)) 3) 1 4)
1   (LIB FPRINTF (RD (OBJ 13)) (OBJ 18))
2   (LIB EXIT (- 1))
3   (GOTO 4)
4   (WR (OBJ 4) (RD (AREF (RD (OBJ 3)) 1)))
5   (WR (OBJ 5) (RD (AREF (RD (OBJ 3)) 2)))
6   (WR (OBJ 6) (LIB FOPEN (RD (OBJ 5)) (OBJ 19)))
7   (GOTO (= (RD (OBJ 6)) 0) 8 11)
8   (LIB FPRINTF (RD (OBJ 13)) (OBJ 20) (RD (OBJ 5)))
9   (LIB EXIT (- 1))
10  (GOTO 11)
11  (CALL GREP (RD (OBJ 4)) (RD (OBJ 6)))
12  (LIB FCLOSE (RD (OBJ 6)))
13  (RET)
)

(procedure GREP (PATTERN FP)
```

```
14  (WR (OBJ 12) (LIB FGETS (OBJ 17) 256 (RD (OBJ 16))))
15  (GOTO (/= (OBJ 12) 0) 16 18)
16  (WR (OBJ 12) (CALL INDEX (RD (OBJ 15)) (OBJ 17)))
17  (GOTO (> (OBJ 12) 0) 19 21)
18  (RET)
19  (LIB FPUTS (OBJ 17) (RD (OBJ 14)))
20  (GOTO 21)
21  (GOTO 14)
)

(procedure INDEX (PATTERN LINE)
22  (WR (OBJ 9) 0)
23  (GOTO 24)
24  (GOTO (/= (RD (AREF (RD (OBJ 8)) (RD (OBJ 9)))) 0) 25 28)
25  (WR (OBJ 10) (RD (OBJ 9)))
26  (WR (OBJ 11) 0)
27  (GOTO 30)
28  (RET (- 1))
29  (RET)
30  (GOTO (AND (/= (RD (AREF (RD (OBJ 7)) (RD (OBJ 11)))) 0)
            (= (RD (AREF (RD (OBJ 8)) (RD (OBJ 10))))
              (RD (AREF (RD (OBJ 7)) (RD (OBJ 11))))))
      31 34)
31  (WR (OBJ 10) (+ (RD (OBJ 10)) 1))
32  (WR (OBJ 11) (+ (RD (OBJ 11)) 1))
33  (GOTO 30)
34  (GOTO (= (RD (AREF (RD (OBJ 7)) (RD (OBJ 11)))) 0) 35 36)
35  (RET (RD (OBJ 9)))
36  (WR (OBJ 9) (+ (RD (OBJ 9)) 1))
37  (GOTO 24)
)
```

4.3. Static Analysis Model

Different inputs to a program may cause it to follow different paths in the program graph. Static analysis cannot possibly examine all such execution paths one by one because there could be infinitely many. Therefore, when a simple data-flow analyzer identifies the definitions reaching for a node, it collects the definitions from all paths, including some infeasible ones. When computing the interprocedural reaching definitions, it does not distinguish the calling site for the procedure. The reaching definitions so computed are correct except that they may contain more definitions than needed.

In modern programming languages, a (variable) name can bind to one or more locations in different scopes. For example, a local variable declared in a recursive procedure has as many occurrences as the number of (run-time) invocations of the recursive procedure. With dynamic allocation, each execution of the allocation statement produces one more instance. These instances are indistinguishable in static analysis.

An array is a variable that stores a sequence of values. It is difficult to distinguish array elements. The typical static-analysis approach is to treat an array as one big variable storing a set of values.

We define a static model for programs to reflect these approximations, so that we can reason about them. We describe the model for the control flow (issues on execution context) in Section 4.3.1 and the model for data (issues on the instances of variables and the elements of arrays) in Section 4.3.2. Next, we describe the model for aliases (issues on pointer computation) in Section 4.3.3. Based on the alias model, we define the data flow model in Section 4.3.4. The data flow model is used in program slicing and well-behavedness verification.

4.3.1. Control Flow Model

APG is basically a program-flow graph. A node may have more than one incoming edge. A procedure may have more than one calling site. When static analysis computes the flow information, it does not distinguish the information from different incoming edges or different calling sites. The failure to distinguish this information leads to the loss of precision of flow analysis. Figure 4.6 illustrates the problem. The program in Fig. 4.6 has two calls to the procedure #strcpy()#. The data flow derived with flow analysis is less accurate. The precise data flow is shown in Fig. 4.6a. The failure to distinguish information from two distinct calls to #strcpy()# joins two dependences, as shown in Fig. 4.6b, leading to a less precise result.

```
main() {
    ...
    strcpy(a, b);
    ...
    strcpy(c, d);
    ...
}

strcpy(s, t) char *s, *t; {
    while (*s++ = *t++); }
```

Previous work has improved the precision of flow analysis. For procedure calls, a simple method is to replace a procedure call with the body of the procedure, although it has the disadvantages of increasing the program size. Horowitz tries to collect the call-site information [Horwitz,dependence,slicing] so that procedures called from different locations will have different contexts for analysis. Wegman [Wegman,constant,conditional] worked on constant propagation with conditional branches; the idea is that if any conditional branches are found to have a constant conditional expression, the search for constants can ignore parts of a program that are never executed. All of these techniques work for the special cases and do not handle the loss of precision in flow analysis. On the contrary, program verification (axiomatic approach) solved the problem using predicates to distinguish groups of paths. Paths in the same group have similar characteristics and can be analyzed together; different groups are separately analyzed.

We use the execution context to expand the program at the procedural level to increase the precision of our analysis, and we use predicates at the intra-procedural level for the same purpose. Expanding the program is not easy; it is difficult to provide the right predicates. Automatic expansion can greatly increase the size of the program.

Fortunately, expansion is not always necessary. The tool can run in automatic mode for the initial analysis. If the initial slice is too large, the analyst can create a detailed analysis by expanding the sliced program by providing the appropriate predicates and then slice again. The following two sections elaborate on how to expand the program at the procedural and intra-procedural levels.

4.3.1.1. Expand at the Procedural Level

In Fig. 4.7, #inc()# has two calling sites in #main()#. The first invocation of #inc()# returns 1 and the second invocation returns 2; #main()# returns 2. Without distinguishing two invocations, the input value to #inc()# is the set {0,1}, and the return value is the set {1,2}.

With contextual expansion using the call-site information, we create two instances of #inc()#: labeled #inc[Node1]()# and #inc[Node2]()# where Node1 and Node2 are the node-ids of the calling node in procedure #main()#. We use array notation; the array name identifies the procedure, and the index identifies the instance of the procedure.

```
main()                              inc(i)
{                                   {
   int  i = 0; /* node 0 */            return i+1;
   i = inc(i); /* node 1 */         }
   i = inc(i); /* node 2 */
   return i;   /* node 3 */
}
```

4.3.1.2. Expand at the Intra-procedural level

Similarly, we can extend the execution context idea to the control flow graph within a procedure. We identify different groups of execution paths. The groups of paths are distinguished by predicates. Suppose we divide the paths passing through node N into k groups, where the i-th path is described by the predicate P_i. $P_1 orP_2 or \cdots orP_i or \cdots P_k$ is *true* so that all execution paths passing through node N are considered. We produce k copies of the node N, labeled $N[P_1], N[P_2], \ldots, N[P_k]$. Then we analyze the expanded program. In flow analysis, the predicate narrows down the flow information.

Consider the programs in Fig. 4.8. We are interested in learning the value of #y# in the last statement. Traditional flow analysis shows that #y# can have a value of 0, 1, or 2. In fact, #y#'s value is 2. Flow analysis cannot generate the most precise solution because it does not distinguish information from different paths; both the true-branch and the false-branch of the if-statements are assumed to be taken. By dividing the execution paths into two cases (specified with predicate B and $notB$), the flow analysis produces the desired solution.

---

y = 0; $\{y = 0\}$ if (B) y = 1; $\{y = 0 ory = 1\}$ if (B) y = 2; $\{y = 0 ory = 1 ory = 2\}$ if (B) y == ?

4.8a Traditional flow analysis

y = 0; $\{B ==> y = 0$and$notB ==> y = 0\}$ if (B) y = 1; $\{B ==> y = 1$and$notB ==> y = 0\}$ if (B) y = 2; $\{B ==> y = 2$and$notB ==> y = 0\}$ if (B) y == ?

4.8b Flow analysis using predicates

Sometimes it is not sufficient to express the state of the execution by using the variables and execution contexts of the program. Auxiliary variables may be introduced to record important information. Figure 4.9 contains a small program swapping the value of two variables #x# and #y#. Auxiliary variables #X# and #Y# are used to record the previous values of #x# and #y# respectively. Without #X# and #Y#, it is not possible to represent the assertion that #y# has the old value of #x# and #x# has the old value of #y#.

$\{x = X$and$y = Y\}$ t = x; $\{x = X$and$t = X$and$y = Y\}$ x = y; $\{x = Y$and$t = X$and$y = Y\}$ y = t; $\{x = Y$and$y = X\}$

4.3.2. Data Model

It is not always possible to distinguish different instances of variables with static analysis. For example, we cannot distinguish the arguments to the main procedure in C. The number of arguments depends on how the program is started and that information is not present in the source program.

We identify all storage locations and statically assign them an identity, called an *object-id*. Storage locations which are not distinguishable in static analysis are given the same identity; e.g., elements of an array, fields of structures, and instances of a local variable in a recursively invoked procedure. The data model caters to all allocated storage in the C programming language (as well as other procedural languages). We find that a compiler allocates storage in the following categories.

(1) Those declared as global variables: e.g., GLOBAL:STDIN in sgrep.c Since global variables have exactly one instance, they are uniquely identified by object-ids.

(2) String constants. C compilers allocate storage for string constants. The rvalue of a string constant is actually the address of the allocated storage.

(3) Those declared as local variables and parameters: e.g. MAIN:ARGC, MAIN:I. These objects may refer to more than one instance of the variables only when recursion occurs.

(4) Those allocated by the run-time support, i.e., arguments to the main procedure call.

(5) Those from explicit memory allocation such as malloc, calloc, etc.

(6) Those from implicit memory allocation such as fopen(), freopen(), fdopen(). fopen() opens the file named by filename and associates a stream with it. On success, fopen(), freopen(), and fdopen() return a pointer that identifies the opened stream. These routines from the stdio library use a shared global variable _iob for the stream.

(7) The return value from each function.

Allocation categories (1) through (3) are very straightforward. Allocation category (4) is associated with the run-time support for the C language. The C run-time support allocates storage for the command line arguments as shown in Fig. 4.10. The command line arguments are collectively assigned object-id 1 because it is not possible to know the number of arguments at static time. The array of pointers to the arguments is assigned object-id 0. Objects in category (5) and (6), which are represented slightly differently, use the node-id to distinguish the implicitly or dynamically created objects from others. Consequently, objects created from the same node have the same object-id. Objects in category (6) are allocated in the stack or in a register temporarily during the return of a function call.

We have described only the data model here. A complete example is given in Section 4.2.5. The translation from C variables to objects is discussed at greater length in Section 5.2.

4.3.3. Alias Model

An *object* contains a block of memory that can be examined and stored into. An *lvalue* in C programs is an expression referring to an object in such a way that the object may be examined or altered. If two or more expressions denote the same lvalue, we say that the expressions are *aliases* of one another. Structures and arrays are *composite* *objects* consisting of one or more variables. An lvalue may be *overloaded* as the same value may represent the composite object or its first component; the type of the lvalue expression disambiguate it.

There are many ways of generating lvalues in C: dereferencing of a pointer variable, applying the address operator (#&#) to a variable, or applying arithmetic operations with lvalues and integers. An lvalue may be returned from a function providing dynamic memory allocation, e.g., malloc(). Sometimes an integer can be type-cast to obtain a lvalue, although the lvalue may not contain a legitimate address. All these lvalues are a generalized version of pointers and hence should be handled for alias analysis.

For simple objects, the lvalue is simply the address allocated for the object. For composite objects, the lvalue designates the base address of the memory allocated for the object, and an offset that identifies the particular element being accessed. When we map allocated addresses into object-ids, we ignore the offset portion of the lvalue.

The effect of pointer assignments in data flow analysis problems has been studied to varying degrees. In the extreme case, this particular problem has been completely ignored. Banning [banning,aliases] has addressed this problem, but his solution is not entirely satisfactory. He does not distinguish among variable locations pointed by different pointer variables of the same type. Moreover, he does not treat the problematic case in which pointer variables point to the same locations to which variables map. Chow [Chow,design,data,flow] has also addressed the pointer assignment problem, but she only handles single-level pointer dereference. With each pointer variable #a#, she associate a target variable #*a# which is the storage location accessible through #a#; the rest of the computation treats #*a# as a variable.

Before describing our analysis, we first introduce the types of aliasing and the aliasing representation in general. We then further discuss these issues as applicable to C programs, and then our alias analysis.

4.3.3.1. Types of Aliasing

There are several kinds of aliasing, listing in increasing order of complexity: static aliasing, reference parameter aliasing, and pointer aliasing.

- **Static aliasing.** Examples include common blocks in FORTRAN, variant records in Pascal, and union structures in C. The aliases can be identified by determining the set of variables sharing storage. The binding can be determined at compile time. The scope of aliasing is the whole program.
- **Reference parameter aliasing.** Reference parameter aliasing occurs in programming languages supporting call-by-reference parameter passing. The aliasing holds for the scope of a procedure. Aliasing information flows from the calling procedure to the called procedure, but does not flow in the other direction. Static determination is easy.
- **Pointer aliasing.** Pointer aliasing is the most complicated aliasing problem since pointers can be assigned, modified, computed, and passed as parameters. Call-by-reference parameter passing might be simulated using pointers and call-by-value parameter passing. The aliasing is only valid for the scope of one statement--valid between variable modifications. The worst thing is that aliasing information can flow both forward and backward in the procedure calls. Another important characteristic of pointer aliasing is that the effect of pointer assignment is not localized. With reference parameter aliasing, the aliasing effect is limited to the body of that procedure and probably the called procedures. With pointers, as shown in Fig. 4.11, the pointer assignment can alter the aliasing information back in the calling procedure, in addition to its effect on the called procedure. (Section 4.3.3.4 and Fig. 4.12 explain how that happens.)

```
char d;

main() {          char *p, c;          p = &c;                        /* ptr(p) = {c} */
        subprogram(&p);          putchar(*p);          /* ptr(p) = {d} */ }

subprogram(x) char **x; {          *x = &d; }
```

4.3.3.2. Scope of Aliasing

- **Program-level.** The aliasing information holds for the whole program. Suppose p is a pointer. $lvalue(p)$ is the set of variables that p points to. p may point to different variables in different places in the program. $lvalue(p)$ collects all the variables that p may point to somewhere and sometime in the program without distinguishing the location.
- **Procedure-level.** The aliasing information holds for the scope of a procedure; the information may be different for different procedures. The reference parameter aliasing has this scoping.
- **Statement-level.** The aliasing information holds only for a block of statement that has no gotos, no procedure calls, and no pointer assignments. The pointer aliasing is of this scope.

An alias problem in the statement-level scope can be transformed into more relaxed scoping to obtain a less precise solution. Therefore, we can still solve the pointer aliasing using the program-wide scope if precision is not the prime objective. Nevertheless, such simplification generally produces program slices that are too large to be useful.

4.3.3.3. Alias Representations

This section defines three types of alias representations.

- **The variable equivalence relation model.** Aliasing information is represented as equivalence relations between variables, i.e.,

$$ainalias(a)$$
$$ainalias(b) ==> binalias(a)$$
$$ainalias(b)\mathrm{and}binalias(c) ==> ainalias(c)$$

It handles parameter aliasing and single-level pointer aliasing. It cannot accurately represent the multiple level of pointer indirections. Usually this model is used at the procedure-level analysis. The cost of maintaining the equivalence relation does not allow this model to be used with the statement-level analysis.

- **The pointer mapping model.** Aliasing information is not directly represented. Instead, the contents of pointer variables are traced and stored in a directed graph. The nodes are variables; an edge from a to b represents that pointer a points to variable b. This model can handle general pointer expressions.

  Suppose p is a pointer and env is the mapping from pointer variables to their values at node n. $lvalue(p, n)$ is the set of objects that pointer p points to at node n. $lvalue(p, n)$ is more precise than $lvalue(p)$ because the position of the pointer is known. The explicit use of pointers allows multiple pointer dereferencing to be handled, e.g., the assignment of #**p# at statement n may modify objects in the set $lvalue(lvalue(p, n), n)$.
- **The symbolic model.** This model is an enhancement with the pointer mapping model. It uses a predicate to further discriminate pointer values in different execution paths, i.e., lvalue(p, n, predicate). However, automatic generation of the appropriate predicates is an open research problem.

4.3.3.4. Pointer Mapping Analysis

The problem with analyzing C programs is now to deal with the pointer aliasing problem. Different from typical alias analysis, which performs the analysis with the equivalent aliasing relation at the procedure level, we have to use the pointer mapping model at the statement level. The precision is controlled by adding predicates to the system with the method described in Section 4.3.1.

The analysis is done by tracing the pointer values and storing them in a mapping function. The tracing is executed for each pointer assignment and the parameter bindings at procedure calls. Figure 4.12 illustrates the notion of pointer tracing using the example in Fig. 4.11.

In practice, we associate the mapping, called *pmap*, from pointer objects to a set of object-ids with each node. A pointer object may correspond to more than one pointer variable. In the program in Fig. 4.13, #ap# is an array of pointers: #ap[0]#, #ap[1]#, and #ap[2]#. The elements of the array of pointers are not distinguished in static analysis, therefore one object-id is assigned for them. Suppose the object-id for #ap# is 0, #x# is 1, #y# is 2, #z# is 3, #p# is 4, and #q# is 5. We have different pointer pmaps for each statement, e.g, $pmap_0$, $pmap_1$, etc.

char *ap[3], x, y, z, *p, *q; /* Object id assignment: ap is 0, x is 1, y is 2, z is 3, p is 4, q is 5. */

/* $pmap_0 = \lambda(x)\ empty$ */ p = &x; /* $pmap_1 = pmap_0[4 \rightarrow lb1rb]$ */ ap[0] = &x; /* $pmap_2 = pmap_1[0 \rightarrow lb1rb]$ */ ap[1] = &y; /* $pmap_3 = pmap_2[0 \rightarrow lb1,2rb]$ */ ap[2] = &z; /* $pmap_4 = pmap_3[0 \rightarrow lb1,2,3rb]$ */ q = ap[i]; /* $pmap_5 = pmap_4[5 \rightarrow lb1,2,3rb]$ */ /* $pmap_5 = pmap_0[4 \rightarrow lb1rb, 0 \rightarrow lb1,2,3rb, 5 \rightarrow lb1,2,3rb]$ */

char *x, **y, ***z, a; /* Object id assignment: x is 0, y is 1, z is 2, a is 3. */

/* $pmap_0 = \lambda(x) empty$ */ x = &a; /* $pmap_1 = pmap_0[0 \rightarrow lb3rb]$ */ y = &x; /* $pmap_2 = pmap_0[1 \rightarrow lb0rb]$ */ z = &y; /* $pmap_3 = pmap_2[2 \rightarrow lb1rb]$ */ /* $pmap_3 = pmap_0[0 \rightarrow lb3rb, 1 \rightarrow lb0rb, 2 \rightarrow lb1rb]$ */ a = ’c’; *x = ’c’; **y = ’c’; ***z = ’c’;

We examine the statement #***z=’c’# in the program in Fig. 4.14. The pointer mapping for that statement is $pmap_3$. The object id assignments are: #x# is 0, #y# is 1, #z# is 2, and #a# is 3. Suppose we want to determine which object is modified in the last four assignment statement in that program. We start with the left side of the assignment statement #a=’c’;#, that is #a#. Therefore object 3 is modified. In the assignment statement #*x=’c’;#, $pmap_3(0)$ returns the set of objects $\{3\}$, therefore object 3 is modified.

Statement #**y=’c’# has two indirections, therefore we apply $pmap_3$ twice. The set of objects is shown in the following figure.

$$(pmap_3\ t) wheretin (pmap_3\ 1).$$

Similarly, statement #***z=’c’# has three indirections, we apply $pmap_3$ three times. The set of objects is shown in the following figure.

$$(pmap_3\ t) wheretin (pmap_3\ u) \mathrm{and} uin (pmap_3\ 2).$$

Given the pointer mapping $pmap_n$ at node n, we still need to define a function, called *lvalue*, to compute the left side and the right side of an assignment statement for alias analysis. When *lvalue*, which takes *pmap* as a parameter, evaluates the left side, it determines which pointer object is being modified. When *lvalue* evaluates the right side, it returns the list of object ids, i.e., address of the variables in real execution, if the right side is a valid pointer expression. Figure 4.15 shows how *lvalue* is defined.

<table>
<tr><th colspan="2">(lvalue EXPR pmap)</th></tr>
<tr><th>EXPR</th><th>Result</th></tr>
<tr><td>(OBJ id)</td><td>id</td></tr>
<tr><td>(RD expr)</td><td>(pmap (lvalue expr pmap))</td></tr>
<tr><td>(AREF expr index)</td><td>(lvalue expr pmap)</td></tr>
<tr><td>(SEL expr field)</td><td>(lvalue expr pmap)</td></tr>
<tr><td>(+ expr1 expr2)</td><td>(set-union (lvalue expr1 pmap) (lvalue expr2 pmap))</td></tr>
<tr><td>(- expr1 expr2)</td><td>(lvalue expr1 pmap)</td></tr>
</table>

4.3.4. Data Flow Dependence

The data flow specifies the relationship between variables used and modified. In a simple language without side-effect expressions, pointers, or arrays, the variables on the right side of an assignment are said to be *used*, and the variable on the left side is said to be *defined*. The variable modified is said to be *dependent* on the variables used. It is no more complicated with arrays, assuming no out-of-bound array accesses.

However, it is more complicated with the presence of pointers. We assume the dereferenced pointers refer to some valid variables in the program. If the pointer refers to an array, the element it accesses is assumed not out-of-bound. In an assignment statement with pointers, the objects used and modified are no longer determined in a straightforward way; they are computed using the *lvalue* defined in the alias model.

The correctness of the data flow dependence depends on the validity of the pointer expressions and the absence of out-of-bound array accesses, i.e., the well-behavedness. These properties are assumed and must be validated before we use the data flow dependence as the basis for analysis.

Global data flow analysis of the program is incomplete without knowing how the library calls manipulate the data. We can derive the data flow relation for the input and output of a user-defined procedure in the program, but we have no way to discover it for library calls. Therefore, we use a data flow definition table to supply the information. The library defines the *gen*, *kill*, *use*, and *ret* values of the function. When the value of an object is modified, a *definition* of the object is said to be *generated*. *Gen* contains the set of definitions generated by the library. When an object is modified, either some or all of its previous content is erased. If all of them are erased, previous definitions of the objects reaching the library are no longer valid. *Kill* contains this set of definitions that can be removed. *Use* contains the set of objects referenced by the library. *Ret* contains the possible return value from the library call. The gen, kill, use, and ret information is defined as a function of the formal parameters and global variables. The return value is directly used for data flow analysis, but it is useful for alias analysis, i.e., computing *lvalue*, when the library call may return a pointer.

Figure 4.16 illustrates the data flow definition for the library function #strcat()#. #str1# and #str2# are pointers binding to the actual parameters. The expression `(use (rd (obj str1)))` means the variables pointed to by #str1# are used. Similarly, the expression `(gen (rd (obj str1)))` means the variables pointed to by #str1# are modified. The return value is the object id stored in the pointer #str1#.

```
(deflib strcat (str1 str2)
    (use (rd (obj str1)))
    (use (rd (obj str2)))
    (gen (rd (obj str1)))
    (ret (rd (obj str1)))
```

4.4. Summary

We define a program model, which abstracts the static features of a program, for analysis. The advantage of specifying a static model is that we explicitly specify the assumptions used with static analysis. When the model is correct, then the result obtained is correct, too. The only way to defeat our analysis tool is to create discrepancies between a C program and its abstracted program model on which analysis is being done. It is important to define the program model because not all syntactically correct C programs are well-behaved. By demonstrating the well-behavedness, we eliminate the only way our analysis tool may be fooled into producing incorrect results.

The static model is a finite abstraction of the dynamic and possibly infinite executions of a program. The abstraction allows tractable and usually more efficient analysis to be done, but it may lose certain important information of the program. We translate the whole array into an object to allow automatic data flow analysis to be performed. It is usually very difficult and also useless to distinguish different elements of an array automatically, but the programmer may have some good ways to partition an array into regions to be analyzed. Our model provides a mean for the analyst to carry out these kind of refined analyses by providing a mechanism for the programmer to supply predicates to expand the analyzed program. Therefore, our model is general enough for a range of flow analyses, from fully automated to human assisted analysis. The predicate method was actually borrowed from program verification techniques. At one end of the spectrum, in which the analysis is not automated, our model can support proof as powerful as program verification.

In the software engineering aspect, the APG representation supplies unique numbering for the nodes in the program graph and for the objects. The numbering allows using bit vectors for data flow analysis and pointer mapping analysis. The flow analyzer is simplified because it only needs to work with a simple and well-defined language. The APG form is extensible; more and more information is added to the form during flow analysis. It becomes very easy to integrate the analysis of various modules together by using the APG as the common form for communication.

CHAPTER 5

Translator

5. _5

The translator compiles C programs into APG representation, which is discussed in Chapter 4. Section 5.1 presents an overview of the translator. We assume that the readers are familiar with compiler construction. [Aho,ullman,compiler] Sections 5.2 through 5.9 discuss the more interesting issues in implementing this translator. Section 5.10 summarizes our implementation experience and suggests improvement for the future.

5.1. Overview

The translator makes two passes. The first pass is written in lex [lesk,lex] and yacc [Johnson,Yacc] and generates a symbol table for the variables, a symbol table for type identifiers, and the internal code for the procedures and statements. The second pass is written in Common Lisp [Steele,Common,Lisp] [KCL] and handles the context-sensitive translation rules. It also transforms the internal form to satisfy the syntactic requirement defined in the program model in Section 4.2.

We implemented a subset of the C language. Our grammar is significantly simpler than that for the full C language, but it is sufficient for writing any program. It serves as a prototype to generate an internal form for testing purposes. Features such as typedef, labels and gotos, variable declaration at the beginning of a compound statement, comma-expressions, and usage of function identifier as variable are prohibited in order to simplify our translator. Since MCF needs a complete program for analysis, we do not support separate compilation.

Program transformation in the second pass eliminates problematic cases that occur in the C language, such as the order of evaluation and side effects.

5.2. Variables versus Objects

Let us examine the files sgrep.c and sgrep.i in Figs. 4.4 and 4.5 respectively. Each procedure in the C program has a corresponding procedure in the APG program. Fig. 5.1 illustrates the relationship of C variable declarations to the APG object-id representations. The correspondence is maintained using the 3-tuple: (OBJECT-ID PROCEDURE-NAME VARIABLE-NAME).

C Declarations

```
main(argc, argv) int argc; char *argv[]; {
  char *pattern, *file;
  FILE *fp;
```

(2 MAIN ARGC) (3 MAIN ARGV) (4 MAIN PATTERN) (5 MAIN FILE) (6 MAIN FP)

APG Object ids

The APG must account for all kinds of memory locations as described in Section 4.3.4. Figure 5.2 lists the APG allocations and their corresponding categories. Object 12 (GLOBAL $0) has no corresponding program variables; variables starting with $ are global temporary registers introduced by program transformations. Recall the seven different allocation categories defined in Section 4.3.2:

Allocation Category	**Object-ID**	**C Objects**
C global variables	12	GLOBAL $0
	13	GLOBAL STDERR
	14	GLOBAL STDOUT
C string constants	18	CONST "Usage: sgrep <pattern> <file>\\n"
	19	CONST "r"
	20	CONST "sgrep: cannot open file %s\\n"
C local variables and formal parameters	2	MAIN ARGC
	3	MAIN ARGV
	4	MAIN PATTERN
	5	MAIN FILE
	6	MAIN FP
	7	INDEX PATTERN
	8	INDEX LINE
	9	INDEX I
	10	INDEX J
	11	INDEX K
	15	GREP PATTERN
	16	GREP FP
	17	GREP BUF
Run-time storage allocation	0	CRT0 ARGP
	1	CRT0 ARGS
Dynamic allocation		not used
Implicit allocation	21	POINT 6
Return values	22	MAIN RET
	23	INDEX RET
	24	GREP RET

- C global variables (1) are uniquely identified with the keyword GLOBAL and the variable name.

- C string constants (2) are uniquely identified with the keyword CONST and the constant itself.
- C local variables and formal parameters (3) are uniquely identified with the procedure and variable names.
- Storage locations allocated by the run-time support (4) are uniquely identified with the keyword CRT0 and a label.
- Dynamic allocation (5) and implicit allocation (6) are uniquely identified by the keyword POINT and the node-id of the statement in which they are generated.
- Return values (7) are uniquely identified by the function name and the keyword RET.

5.3. Loop Statements

We use standard compilation techniques to translate C programs to APG code. In the first pass, we generate labels for the branch-condition, the entry of the loop-body, and the exit of the loop-body. In the second pass, we patch the labels with actual node-ids. Figure 5.3 show the translation of the while-statement; the translations for other loop constructs are similar. Yacc provides a mechanism to pass inherited attributes [Knuth,attribute] from the tokens on the left side to those on the right side; manipulation of the parser stack is necessary to pass synthesized attributes [Knuth,attribute] from right to left. Even so, the synthesized attributes must be computable in one pass.

We reduce the use of synthesized attributes by generating labels for the while-statement at the left side and passing them to the right. Three labels are generated to indicate the starting point of the boolean expression, the beginning of the while-body, and the end of the while-statement. Figure 5.3 shows the Yacc code. The first label is generated after seeing the WHILE token and is emitted immediately after being generated. The second label is generated and also emitted after accepting the boolean expression. The third label is generated after accepting the boolean expression, but it is passed to the end of the body and emitted there.

```
C statement:                while (a < 0) a++;

Code after 1st pass:              L001:                (GOTO (< (RD A) L002
L003)               L002:                (WR A (+ (RD A) 1))
(GOTO L001)              L003:

Code after 2nd pass:              0   (GOTO (< (RD (OBJ 0)) 0) 1 3)              1
(WR (OBJ NIL) (+ (RD (OBJ 0)) 1))              2   (GOTO 0)

Yacc:   while-statement                 :  WHILE                    {   loopsave();
                 $$ = genlabel();/* first label */                 loopcont = $$;
                 emitlabel($$);                  }                '(' expression ')'
```

```
        {               char   *L1;             L1  =  genla-
bel();        /* second label */          $$ = genlabel();         /* third
label */            loopbreak = $$;              sprintf(buf, "(goto %s %s
%s)",  $4,  L1,  $$);              emitcode(buf);                emitlabel(L1);
        }             statement            {                  sprintf(buf,
"(goto %s)", $2);          emitcode(buf);
        emitlabel($6);              looprestore();            }
```

A while-statement can contain continue or break statements. The continue-statement starts another iteration of the smallest enclosing loop, i.e., goto the first label. The break-statement passes control out of the smallest enclosing loop, i.e., goto the third label. We use two global variables, #loopcont# and #loopbreak#, to remember the labels. Since loops can be nested, we use the procedures #loopsave()# and #looprestore()# to preserve the continue and break information on a stack, simulating synthesized attributes. The function #emitcode# produces internal statements.

With this translation scheme, we need a label for both the beginning and the end of a loop. A location between two consecutive loops will have two distinct labels--one is the end point of the earlier loop and the other is the starting point of the later loop. One function of the second pass is to eliminate the duplicated labels, mainly for cosmetic purposes.

5.4. Order of Evaluation

In the C language, [Kernighan,Ritchie,C] the order of evaluation of expressions is defined by the precedence and associativity of the expression operators. Otherwise it is undefined. In particular, a compiler is free to generate code that computes subexpressions in any order following the precedence and associativity constraints, even if the subexpressions involve side effects. The order in which side effects take place is therefore unspecified. Expressions involving a commutative and associate operator (*, +, &, |, ^) may be rearranged arbitrarily, even if used with parentheses. Figure 5.4 shows a program with two procedures illustrating the problem. The expression #f()+g()# yields two different values depending upon which one is executed first. When #f()# is evaluated first, the expression's value is 1. When #g()# is evaluated first, the expression's value is 2.

```
int x = 0;

f() {
```

```
         if (x) return 0;
         x++;
         return 1; }

      g() {
         if (x) return 0;
         x++;
         return 2; }
```

Although the different evaluation orders of a side-effect expression may be handled with a non-deterministic model, we think the effort of creating this model far outweighs its usefulness. Our translator generates code that uses strict left-to-right order for commutative operators and actual arguments in a function call. Therefore, the side effects associated with the subexpressions are also evaluated left-to-right.

5.5. Elimination of Side Effects

Expressions containing side effects do not only compute a value, but they also modify some variables. Some examples are #var++#, #var−−#, #++var#, #−−var#, and functions that modify non-local variables. We eliminate side effects, except the variables modified in a procedure call, during the translation by using explicit temporary registers to force a strict left-to-right order of evaluation. Figure 5.5 shows the translation scheme. #alloc-reg()# is used to allocate temporary registers for storing the values of the expressions. The registers are freed when a statement ends.

```
Statement:                    x = i++ + i;

Object Allocation:                  Register = (OBJ 0)                    x = (OBJ 1)
                  i = (OBJ 2)

Internal Code:                 (WR (OBJ 0) (RD (OBJ 2)))                        (WR
(OBJ 2) (+ (RD (OBJ 0)) 1))                (WR (OBJ 1) (+ (RD (OBJ 0)) (RD (OBJ
1))))

Yacc: expression                   : "++" lvalue                          { char *reg;
                        reg = alloc-reg();                            sprintf(buf, "(wr
%s    (+    (rd    %s)    1))",    reg,    $1)                           emitcode(buf);
                        sprintf(buf, "(wr %s %s)", $1, reg);
emitcode(buf);                              $$   =   reg;                             }
                  |      lvalue      "++"                     {      char      *reg;
```

```
                        reg = alloc-reg();                                 sprintf(buf, "(wr
%s      (rd      %s))",       reg,      $1);                                  emitcode(buf);
                           sprintf(buf,     "(wr     %s     (+     %s     1))",    $1,     reg)
                          emitcode(buf);                                     $$    =    reg;
                        }
```

5.6. Patching the Control Flow

A C procedure returns implicitly when the execution finishes the last statement in the procedure. We make the return explicit by adding a return statement to the end of the procedure. The explicit return is useful when a procedure has multiple exit points: we have to combine the flow information from all exit points, including the implicit return at the end of the procedure.

We may use #exit()# to terminate a program. The first pass of the translator recognizes #exit()# as a normal procedure call and generates an edge from the node containing the #exit()# call to the successor node. The second pass removes this control-flow edge so that data-flow analysis terminates at the exit node. Therefore, if a procedure exits and never returns, its data-flow information will not propagate to the calling procedure.

5.7. Procedure Calls Transformation

We distinguish between user-defined procedure and library calls. The data flow of the former must be determined by the flow analyzer, but that of the latter is supplied from the library declaration. The distinction is not possible until the whole program is input. In the first pass, we use the CALL instruction to denote both user procedure calls and library calls. In the second pass, we patch the internal code to use CALL for user procedure calls and LIB for the library calls.

We do not allow more than one procedure call in one node for APG in order to make the implementation of the flow analyzer easier. The reason is that the pointer-mapping function can change between procedure calls, and we only want to associate one mapping function with one node.

The flow analyzer has to keep track of the uses and definitions of the variables in the called procedures. We use explicit temporaries to break down the statement so that each node will have at most one procedure call. Figure 5.6 is an example of the transformation. First, the intermediate code is split into two lines. A register (OBJ 1) is introduced to store the temporary result. Second, the CALL to #strlen# is changed to LIB.

The transformation is not restricted to intermediate code with nested procedure calls. We also apply the same transformation to remove procedure calls from expressions.

```
C statement:                    write(1, str, strlen(str));

APG after 1st pass:                      (CALL WRITE 1 (OBJ 0) (CALL STRLEN (OBJ
0)))

APG  after  transformation:                      (WR  (OBJ  1)  (LIB  STRLEN  (OBJ  0)))
                  (CALL WRITE 1 (OBJ 0) (RD (OBJ 1)))
```

5.8. Type-Driven Transformation

The translation from C to APG is not context free. The main problem is that the code generated for simple variables and array variables is different. Let us examine the program in Fig. 5.7. The variable #x# in both statements 3 and 4 obtains the same value. The parse tree for the expression #arr[3]# and #p[3]# is the same. However, the code generated is different because #p# is a pointer and #arr# is an array. The following documents this difference:

- The expression #arr# at statement 3 is translated to (OBJ 1).
- The expression #p# at statement 4 is translated to (RD (OBJ 2)).

```
C  program:                    1    char  arr[10],*p,  x;                    2    p  =  arr;
                  3   x = arr[3];                     4   x = p[3];

APG  program:                   (Variables                          (1  GLOBAL  ARR)
                   (2    GLOBAL    P)                               (3    GLOBAL    X)
                  )                                                  (procedure         ....
                  2  (WR  (OBJ  2)  (OBJ  1))                   3  (WR  (OBJ  3)  (RD
(SEL  (OBJ  1)  3)))                   4  (WR  (OBJ  3)  (RD  (SEL  (RD  (OBJ  2))  3)))
                  )
```

Our C translator first translates these kinds of expressions into the form (RD (SEL (RD* (OBJ _) 3))). RD* is a special operator and its semantics depend on the type of the expression. RD* is defined as:

(RD* X) → X if the X is
(RD* X) → (RD X) otherwise.

The type analysis of the expression is defined recursively on the types of the subexpressions. Some complicated expressions such as #a->b[5].c->d# are allowed.

5.9. Indirect Procedure Calls

The address of a procedure can be assigned to a pointer in C, and the procedure may be called indirectly through the pointer. The indirect procedure call causes problems in data-flow analysis because we no longer have a static call graph to work with. If the analysis assumes the worst case (i.e., any procedure may be called by an indirect procedure call) the resulting analysis may lose too much information.

Although our prototype does not allow indirect procedure calls, this feature can be easily added to the translator. A procedure pointer differs from other pointers. It cannot have any computations (additions or subtractions). Only assignment is allowed, and the value assigned must be the entry address of a procedure in the program. We can easily use the pointer-mapping analysis to determine exactly which procedure can be called.

If the pointer points to more than one procedure, we need to assume the worst case by combining the flow from these procedures. We argue that this combination does not lose too much information because the procedures that could be called indirectly through the same pointer variable should perform similar function and exhibit similar input-output characteristic. For example, a function pointer passed to the UNIX #qsort()# library routine can point to different comparison functions (e.g., string-compare, float-compare, integer-compare).

5.10. Discussion

We implemented a prototype for our analysis tool. Because it is a prototype, our translator is not implemented for the full C language. In writing the LALR grammar, we found that implementing certain features was complicated and that we needed to use some tricks.

We used a one-pass compiler architecture to generate code from C source programs; a one-pass compiler is simpler than a multiple-pass compiler, although it cannot handle the features that are best defined with synthesized attributes. [Bochmann,semantic,evaluation] We found the use of a one-pass compiler for the subset C language difficult; and for the full language would be impossible.

The C language allows explicit type conversions. The syntax for type conversions is very similar to the syntax for expressions. The following shows one example that is difficult to handle using LALR grammar.

```
/* id is a type name */              /* id is a variable */
typedef int id; int x;               int id;

x = (id) 10;                         x = (id);
```

An LALR grammar may have reduce-reduce conflicts when two production rules have common prefixes, such as the rule for type conversion and parenthetic expressions. The Yacc-generated parser fails to accept these production rules. We need to modify the lexical analyzer to separate identifiers into two groups: typenames and variable identifiers. They must also return TYPENAME and IDENTIFIER tokens.

The comma token in the C language is used for two purposes: the comma expression and the function argument separator. A comma expression is a list of expressions separated by commas; it is evaluated left-to-right, and its value equals that of the rightmost expression. The comma expression can be accepted with LALR grammar by defining a non-terminal called #expression_list#, which is a list of expressions separated by commas. If the expression list is a comma expression, we generate code for each expression and return the value of the last expression. If the expression list contains arguments for a function, we maintain a list of the values for all the arguments that will be emitted together with the CALL instruction later. The use of a one-pass compiler makes it difficult to generate different codes for an expression list. The following shows examples where determining comma expressions is difficult.

```
/* Examples using the , token */
f (a, b);          /* call function f with values of a and b */
f (c, (a, b));     /* call function f with values of c and b */
x = (a, b);        /* assign value of b to x */
```

C's assignment expression has expressions on both the left and right side. The semantic action for the expression on the left side of an assignment expression is different than that on the right. Suppose we ignore the restriction, which requires a valid address, for the expression on the left side and specify the grammar as follows:

```
assignment:   expr '=' expr;
expr:         identifier;
```

In this case, the code generated for the assignment expression #A=A;# is (WR A (RD A)). The IDENTIFIER #A# is reduced to the expression token on both sides. The problem is that for the same reduction, two different values are returned, i.e., A and (RD A). We solve this problem by redefining our grammar to have an *lvalue* expression, so that defining different actions is possible.

```
assignment:   lvalue '=' expr;
lvalue:       identifier;
expr:         identifier;
```

We used a one-pass compiler to generate our internal code because it was easier to implement than a multiple-pass compiler. We believe that a one-pass compiler is only suitable for implementing simple programming languages and recommend a multiple-pass compiler to handle more complicated languages (e.g., the full C language). As described above, one-pass compiler has some limitations, such as the comma and the assignment-expression problems. Although we handled assignment expression in our translator by expanding the production rules, the grammar may not always be LALR after expansion. These two problems are much easier to solve if the first pass generates the parse tree and the second pass produces code from the parse tree. Furthermore, a second pass also handles the type-driven transformation described in Section 5.7.

An alternative to this solution is to use an attributed-grammar parser-generator tool. In this case for the comma expression, we would pass inherited attributes to the code generation to determine whether the current list of comma-separated expressions is a list of arguments or just comma expressions, so that different code could be generated. For the type-driven transformation, we would synthesize the type of the expressions so that the parser can decide what to generate.

CHAPTER 6

Program Flow Analyzer

6. _6

Recall from Section 3.1 that program analysis can be divided into several disciplines: syntax analysis, flow analysis, symbolic execution, and program verification. In this chapter, we focus our discussion on flow analysis. In particular, we are most interested in forward-flow problems.

Section 6.1 defines the relevant terms. Section 6.2 describes a simple intra-procedural flow-analytic framework. Section 6.3 compares and contrasts two approaches to perform inter-procedural analysis. Section 6.4 implements our program flow analyzer by combining one of the inter-procedural approaches with the flow-analytic framework. Section 6.5 expresses in this new framework several flow-analysis problems--pointer-mapping, reaching-definition, node-dominance, acyclic-reaching-definition, and integer-range analysis. Section 6.6 discusses the implementation aspects of flow analysis.

The program flow analyzer works on the intermediate form program and computes the information used by the program slicer and well-behavedness checker. Therefore, we frequently refer to the program model defined in Section 4.3 and make forward references to parts of Chapters 7 and 8 that use the flow information.

6.1. Definitions

A *flow* *problem* involves the local collection of information distributed throughout a program. [Kam,framework] [Kaplan,types] [Kildall,global] [Cousot,abstract,interpretation] Flow problems constitute a class of problems that can be solved in essentially the same manner using identical algorithms.

Flow *analyses* are the techniques to solve flow problems. Typical flow-analysis algorithms work with a simple hypothetical language (e.g. program graph) and ignore procedure calls. These algorithms perform *intra-procedural* flow analysis. Nontrivial extensions are required to apply the algorithms to a more sophisticated language that includes procedure calls and parameter passing. These extensions are called *inter-procedural* flow analyses.

Flow problems can be divided into forward-flow problems and backward-flow problems. A *forward-flow* *problem* involves propagating information along execution paths. The flow analysis determines properties that can reach a certain point in the program. A *backward-flow* *problem* involves propagating information backwards through execution paths. The flow analysis determines the properties expected at a certain point based on subsequent computations, but does not determine properties useful in the security-related analysis. For example, in an array reference, the subscript expression is expected to be within the bounds of the array, but it may still be out-of-bound.

Program graphs often fall into a category called *reducible* [Aho,Ullman,compiler] graphs. A flow graph is *reducible* only if we can partition the edges into two disjoint groups: *forward* and *back* edges which have the following properties:

(1) The forward edges form an acyclic graph in which every node can be reached from the initial node of the graph.

(2) The back edges consist only of edges whose heads dominate their tails. If $a \rightarrow b$ is an edge, b is the head and a is the tail. b dominates a if every path from the initial node of the flow graph to a goes through b.

Informally, a program that has no jumps into the middle of loops from outside and has exactly one entry into each loop has a reducible flow graph.

6.2. Intra-procedural Flow Analysis

The solutions to intra-procedural flow problems can be divided into the interval and the iterative approach. The *interval* approach [Cocke,global,1970] [Kennedy,global,flow,1971] [Ullman,fast,algorithms,1973] [Graham,Wegman,1976] collects information by partitioning the flow graph of a program into subgraphs called intervals. Each interval is replaced by a single node summarizing the local information within that interval. The interval algorithm continues to define such interval partitions until the graph becomes a single node itself, at which time global information is propagated locally by reversing the reduction process. The *data-propagation* approach, also called the *iterative* approach, propagates information along the edges in the program graph until all the required information is collected; that is, until the information does not change any more. The algorithm described in Section 6.2.1 is an iterative approach.

The interval approach was generally believed to be faster than the iterative approach. However, a finding [Ullman,Hecht] shows that the iterative approach is, in fact, comparable to the interval approach under certain conditions. The following are the complexities of different approaches, assuming flow data can be manipulated in constant time:

- The complexity of Allen's interval approach [Allen,data,flow,1974] is $O(ne)$, where n is the number of nodes and e is the number of edges in a program graph. The program graph must be reducible. [compiler,ullman,aho]
- The complexity of Graham and Wegman's interval approach [Graham,global,flow] is $O(e \log e)$, where e is the number of edges. The program graph must be reducible and each interval must have a single exit.
- Hecht and Ullman's iterative approach is $O(Dn)$, where D is the interval depth of and n is the number of nodes in a program graph. The program graph must be reducible and nodes are examined in the depth-first order (see Section 6.2.2 for details).

The interval approach only works with reducible graphs. The complexity of the iterative approach is comparable to that of the interval approach in reducible program graphs, but the iterative approach also works on any program graph without modification. Therefore, we use Hecht and Ullman's iterative approach for our flow analyzer for its simplicity and generality.

The interval approach cannot process arbitrary flow problems (e.g., the pointer-mapping problem). Recall the pointer-mapping analysis in Section 4.3.3.4. The computation of the pointer-mapping function *pmap* involves the use of *lvalue*, where *pmap* and *lvalue* are mutually recursive. We cannot summarize the GEN and KILL of pointer-mapping in an interval into constant expressions. Similarly, we cannot express the linear constraints for an interval in the integer-range problem conveniently.

Other than its simplicity, the iterative approach handles any flow data and transfer functions. In the case of pointer mappings where a fixed point solution of *pmap* and *lvalue* is required, the iterative approach implicitly absorbs the fixed-point computation into its iterations.

6.2.1. Flow-Analytic Framework

Flow analysis research focuses on finding a good algorithm to solve flow problems. With the use of the iterative approach, many researchers [Kam,framework] [Cousot,abstract] observed that the skeleton of iterative algorithms used to solve many different flow problems are, in fact, identical. They formalized their observations and developed a framework to solve general flow problems, provided that the flow problems can be expressed in a predefined format, which is described in the following:

- Specify the data type V for the flow data (the set of values propagated).
- Specify a set F of transfer functions of type $V \rightarrow V$. f, an element of F, determines how a program computation changes the flow data.

- Specify a confluence operator to combine flow data when execution paths merge.

We implement the forward-flow analytic framework with the iterative algorithm shown in Fig. 6.1. We also translate the flow problem in the predefined format into an executable representation, as follows:

- Declare the array *in* and *out*. Each array element corresponds to a node in the program graph and contains elements of type V.
- Supply a routine, %f%, that takes a node B and a value v in V. %f% first produces a transfer function for B, applies it to v, and then returns the result.
- Supply a routine, %j%, to apply the confluence operator to a list of values in V.
- Supply a routine, %d%, to tell whether two values in V are different.

```
/*  M is the number of nodes in the program */ flow_data_type  in[M], out[M],
new_in, new_out;

out[B] = top; changed = 1; while (changed) {
    changed = 0;
    for each block B, in depth-first order {
          /* collect entry information */
         new_in := %j% (out[P] for all predecessors P of B)

          /* compute exit information */
         new_out := %f% (B, new_in);

          /* update the out info */
         if (%d% (new_out, out[B])) {
             node_out[B] = new_out;
             changed++;
         }
    } }
```

Before the iteration, the algorithm orders the nodes in a depth-first order. (The reason for such ordering is given in the next section.)

In each pass, the algorithm first collects for each node the flow data from its predecessors and combines them. Second, it applies the transfer function generated by %f% to the merged flow data. Third, if the new flow data is different from the old generated in the previous pass, another pass is required. If no flow data changes, the flow analysis is completed.

6.2.2. Depth-First Ordering

With iterative approaches, we use the depth-first ordering. [Hecht,Ullman] [Knuth,empirical] Here we summarize Hecht and Knuth's results; the interested reader should see the original papers for details.

The naive iterative approach has a worst-case complexity $O(n^2)$. By applying depth-first ordering on the evaluation of nodes in the iterative approach, Hecht and Ullman [Hecht,Ullman] showed that the complexity can be reduced to $O(Dn)$, where D is called *depth*. Given the depth-first spanning tree for the graph, the depth, D, is the largest number of retreating edges on any cycle-free path, i.e., $max(\#(retreat\_edges(any\_cycle\_free\_path)))$. An edge (a, b) is *retreating* when the depth-first number of b is less than that of a.

The depth-first ordering is created by starting at the entry node and searching the entire graph, visiting nodes as far away from the entry node as quickly as possible, illustrated in Fig. 6.2. In Fig. 6.2, the dotted edges signify retreating edges. The depth-first sequence is START, 0, 1, 2, 3, 4, 5, END. That is, initially the nodes are numbered in depth-first order. The depth of the program graph is 2.

Hecht and Ullman [Hecht,Ullman] have also showed that for reducible flow graphs the data-propagation approach is at least as good as interval algorithms. The number of passes needed to propagate any reaching definition along any acyclic path is no more than one greater than the number of retreating edges, D. One more pass is needed to check that all definitions have been propagated through the graph, so the upper bound of passes is $D + 2$.

While Hecht and Ullman investigated the relation between depth and the complexity of iterative approach, Knuth performed an empirical [Knuth,empirical] study on the interval depth, d, of programs. He showed that the typical program graph has very low (average of 2.75) interval depth. The *interval* *depth* of a program is the number of times we must apply interval partitioning with the interval approaches and is always greater or equal to its depth (i.e., $d \geq D$). Therefore, Knuth's result may be used to estimate an upper bound on the number of passes required for applying iterative approaches on forward-flow problems[1] with single-procedure programs. The empirical average is less than 4.75.

---

[1] Depth-first order is advantageous only for forward flow problems. For backward flow problems, we should use the depth-first ordering of the reverse program graph produced by reversing the edges of the program graph.

6.2.3. Example

The following example that shows the computation of available expressions using the intra-procedural framework. An expression such as #A+B# is *available* at a point p in a flow graph if every sequence of branches that the program may take to p causes #A+B# to be computed after the last computation of #A# or #B#. If we can determine the set of available expressions at entrance to the nodes of a flow graph, then we would know which expressions have already been computed prior to each node. Thus, we may be able to eliminate some redundant computation. The following defines the available-expression flow problem.

- Let $K(x)$ be the set of expressions that are *killed* in node x. Informally, expression A*op*B is killed if either A or B is defined within node x. (*op* is an generic binary operator.)
- Let $G(x)$ be the set of expressions that are *generated* in node x.
- Let $AEin(x)$ and $AEout(x)$, for each node x, be respectively the set of expressions available at entrance to and at exit from node x.

The fundamental relationships that enable us to compute $AEin(x)$ for each node x are:

(AE1) n is the entry node. $AEin(n)$ = union of $G(x)$ for all nodes.

(AE2) For x not the entry node, $AEin(x)$ is the intersection of $AEout(y)$ over all predecessors y of x.

(AE3) $AEout(x) = (AEin(x) - K(x)) \bigcup G(x)$, for each node x.

(AE4) Since AE1-AE3 do not necessarily have a unique solution for $AEin(n)$, we want the largest solution, i.e., containing all available expressions.

This problem fits in the flow-analytic framework. We can use the algorithm in Fig. 6.1 to solve the problem by defining the required data type and functions %j%, %f%, and %d% as follows:

- V is the set of available expressions; the *top* element is the universal set. *in* and *out* are arrays containing elements of type V.
- %j% is set-intersection.
- **f**$(node, in)$ is defined as $(in - K(node)) \bigcup G(node)$.
- %d% is set-unequal.

6.3. Inter-procedural Flow Analysis

The classical flow-analysis technique used to analyze procedureless programs transforms programs into flow graphs [Allen,flow,analysis,1974] and assumes that all paths in the graphs can represent actual executions of the program. The classical technique faces several problems with procedures.

Inter-procedural flow cannot be treated as a simple extension of the classical approaches because it calls for a more complicated model, including features such as local variables and parameter passing. Sharir and Pnueli [Sharir,Pnueli] identified two approaches for performing inter-procedural data-flow analysis. The first solution, the *functional* approach, views procedures as collections of structured program blocks and aims to establish input-output relations for each block. We then interprets procedure calls as "super operations" whose effect on the program status can be computed using those relations. Therefore, the effect of a procedure call can be represented as one node in the program graph. This approach relates to the reverse-invocation-order technique [Allen,1974] and most of the known inter-procedural techniques for program verification. The second solution, the *call-strings* approach, [Sharir,Pnueli] blends inter-procedural flow analysis with the analysis of intra-procedural flow, and in effect turns a whole program into a single flow graph. We present more details about the two approaches in the next two sections.

6.3.1. Functional Approach

The functional approach summarizes the side-effects of a procedure on the flow information. Let us use the example in Section 6.2.3 again to show how available expression can be computed with procedure calls, where parameters are passed by reference (assuming no pointers for the moment). To determine which expressions are killed, we must determine when a variable is defined. We use the variable *change* to summarize the side-effects introduced by procedure calls.

We define for each procedure p a set $change(p)$ containing the set of global variables and formal parameters of p that might be changed. Let $def(p)$ be the set of formal parameters and globals having explicit definitions within p. Suppose p calls q. We define:

$$change(p) = def(p) \cup P(p) \cup G(p)$$

where $P(p)$ is the set of global variables or formal parameters of p that are passed to and modified in q. $G(p)$ is the set of global variables changed in q.

Notice that if procedure p makes no procedure calls, then

$$change(p) = def(p)$$

Consider the example in Fig. 6.3. $change(q) = \{u, v, c\}$ where $def(q) = \{u, v, c\}$. $change(p) = \{a, b, c\}$ where $def(p) = \{a\}, P(p) = \{b\}$, and $G(p) = \{c\}$.

```
var a,b,c,d:integer;

procedure q(var u,v,w:integer)
begin
   u := 2;
   v := 3;
   c := 4;
end;

procedure p()
var y:integer;
begin
   a := 1;
   q(y,b,d);
end.
```

If the procedures in a program are not mutually recursive, the best ordering for computing *change* is to start with leaf procedures. Next, we can compute *change* for those procedures calling procedures whose *change* has been computed. This ordering is called the *inverse-invocation* order.

After we use inter-procedural analysis to determine the *change* information, we compute $K(x)$, which is the set of expressions that are killed in calling node x. Expression A*op*B is killed if either A or B is in $change(p)$ where p is called in node x. Hence, the effect of the procedure call can be summarized and localized.

The final step is to run intra-procedural analysis, as described in Section 6.2.3, for each procedure of the program.

Notice that if the procedures are mutually recursive, we still need to use iterations to compute *change*. For the inverse-invocation order to work, *change* must involve unidirectional propagation of information from called to calling procedures.

6.3.2. The Call-Strings Approach

The call-strings approach blends the intra- and inter-procedural flows together. The flow problem is solved using a slightly modified iterative algorithm in which a procedure call is translated into a control-flow edge to the entry point of the called procedure and into one edge from each of its exit point.

Let us continue to use the example in Section 6.2.3. Since now one node in procedure q can affect another in procedure p if p calls q, we need to redefine $K(x)$. Informally, expression A*op*B is killed if either A or B or aliases of A or B (introduced by reference parameters) are defined within node x. We then run the intra-procedural analysis to determine the available expressions.

In the functional approach, each procedure is analyzed exactly once. In the call-strings approach, several passes are needed until the data-flow information does not change. The basic block containing the procedure call can be imagined to be split into two nodes in the program graph, shown in Fig. 6.4. The first node has an outgoing edge to the entry point of the called procedure. The second node has incoming edges from the exit points of the called procedure.

6.3.3. Functional versus Call-Strings

The functional approach is simpler to define and implement and has more efficient implementations for some flow problems than the call-strings approach. The first pass of the functional approach processes the inter-procedural information, and the second pass analyzes the procedures individually. Suppose we use an $O(m^2)$ data-flow algorithm on a program with p procedures, and each procedure has n nodes. The functional approach typically has the complexity $O(p + pn^2)$, where $O(p)$ comes from the first pass and $O(pn^2)$ comes from the second pass.

The call-strings approach integrates the inter- and intra-procedural analysis. However, its computational complexity, $O((np)^2)$, is p times greater than the functional approach. In the call-strings approach, the whole program is analyzed in each iteration, in contrast to the functional approach, in which only one procedure is analyzed. Despite this additional computational complexity, we use the call-strings approach to implement our program flow analyzer because it is simple and can conveniently solve many flow problems that the functional approach cannot.

Specifically, recall the characteristics of pointer-mapping analysis in Section 4.3.3. We cannot define the effect of a procedure on pointer mappings without specifying the input: a pointer mapping may vary depending upon the values of global variables and parameters. Second, analyzing the procedure in the inverse invocation order does not work because alias information can flow both forward and backward through the procedure invocation.

The integer-range problem does not fit into the functional approach very well. The output integer-range is a complicated function (i.e., systems of linear equations) on the input integer-range. The computation of that input-output relation may take more time than simply performing iterative flow analysis on the called procedure.

Conclusively, the flow problems we need to solve are too “hard” to be solved using the functional approach, which is only suitable for a subclass of flow problems having:

- Uni-directional flow across procedural boundaries
- Input-output expressed as a simple function.

6.4. Inter-procedural Flow-Analytic Framework

We combine the flow-analytic framework with the call-strings approach. The intra-procedural flow-analytic framework discussed in Section 6.2.1 requires the type of flow information *flow_data_type* and the three functions %f%, %j%, and %d%. To extend it to handle procedures, we must (1) add the %call% function to handle procedure parameter passing, (2) modify the %f% function to handle return values, and (3) modify the iterative algorithm to include %call% and %f%. To distinguish this algorithm from the previous one, we rename %f% to %code%, %j% to %join%, and %d% to %diff%. The iterative algorithm is defined in Fig. 6.5, and the functions are defined as follows:

```
/* M is the number of nodes in the program graph */ f_type  in[M], out[M], new_in,
new_out, return_value;

out[B] = top; changed = 1; while (changed) {
    changed = 0;
    for each block B, in depth-first order {
            /* collect entry information */
        if B is not an entry node of a procedure
            new_in := $\bigcup$ out[P] where P is a predecessor of B
        else
```

```
        new_in := in[B];

          /* apply transfer function */
      if B has a procedure call with entry point PE {
              /* B contains a procedure call */
          (parameter, return_value) = %call% (B, new_in);
          parameter = %join% (in[PE], parameter);
          if (%diff% (parameter, in[PE])) {
              in[PE] = parameter;
              changed++;
          }
          new_out = %code% (B, return_value);
      } else {
              /* B contains no procedure call */
          new_out = %code% (B, new_in);
      }

          /* update the out info */
      if (%diff% (new_out, out[B])) {
           out[B] = new_out;
           changed++;
      }
  } }
```

- %code% is the transfer function for a line of the intermediate code. It defines the abstract semantics of the language with respect to the flow problem.
- %call% defines the abstract semantics for procedure call, the values returned, and parameter passing with respect to the flow problem. It produces two values: the new parameters and the return-value of the function. Although the return-value is not used in many flow problems, it is significant when the flow information can be returned, e.g., pointer-mappings and integer-ranges.
- %join% is the confluence function that combines information from predecessors.
- %diff% determines whether the flow information is different.

A node that does not contain procedure call is handled in the same way as the simple iterative algorithm. A node that contains a procedure call is handled in three steps. First, the information stored in the actual parameters relates to the formal parameters at the entry point of the called procedure. Second, the information propagates through the procedure and reaches the exits. Third, the information at the exits are collected and propagates to the calling node as the return-value.

Our order of computation is not optimal. Each level of nested procedure invocations requires an extra pass. The ideal solution is to process a calling node twice: once before the analysis of the called procedure to compute the parameter information, and once after the analysis to collect the return values. We can split the call node into a call-begin node and call-end node to achieve the same effect without modifying the iterative algorithm.

With our implementation, the data *in* and *out*, and the functions %code%, %call%, %diff%, and %join% are pointers to the data structures and functions defined with respect to flow problem. All flow problems share the same analysis algorithm so that we only need to develop the algorithm only once.

6.5. Forward-Flow Problems

The following five subsections present the solutions to the global flow problems used in our static analyzer. They are all organized into four similar parts. The first part presents a flow problem. The second part describes the preprocessing and the definition of the supporting functions. The third part discusses information gathering. Finally, the fourth part provides the information for the extension of APG.

6.5.1. Pointer Mapping

6.5.1.1. Description of Problem

The problem is to compute the pointer mappings for the entry and exit points of each node in a program graph. This computation follows the alias model discussed in Section 4.3.3. The APG intermediate form provides the *lvalue* function to compute the lvalue and the rvalue of an assignment statement. The lvalue decides which variables will be modified. The rvalue signifies the addresses of some variables to be stored in a pointer variable.

The *lvalue* function takes the expression to be evaluated and the current pointer mapping. It assumes that the pointer mapping is not modified during the evaluation of the expression, which accounts for our introduction of the syntactic constraints (see Section 4.2.1) on APG. The preprocessing for the syntactic constraints has been completed before translating a program into APG.

6.5.1.2. Preprocessing

The mapping of a pointer is represented with a list, e.g., (x a b c ...), where x is the pointer object and a, b, and c are the objects being pointed to. The *in* and *out* values are sets of pointer mappings, e.g., ((1 3 4) (4 2 3) (5 1) (8 9 0)). The *top* value is the empty list, i.e., ().

6.5.1.3. Solution

The transfer function %code% can be divided into four cases:

- Assignment statements have the form (WR *lexpr rexpr*). We use *lvalue* to compute the object-ids returned with the evaluation of *lexpr* and *rexpr*. Both *lexpr* and *rexpr* can return multiple object-ids because of pointer aliasing. Consider the C statement #*p='c';#. The #*p# part refers to object 1, 2, or 3 if the *in* mapping is (p 1 2 3). The new mapping generated, *gmap*, is the dot-product of *lvalue*(*lexpr*) and *lvalue*(*rexpr*). When *lvalue*(*lexpr*) contains one simple object of pointer type, the previous mapping of that pointer can be erased. After the erasure, the *out* mapping is produced by merging the *in* mapping with the generated mapping *gmap* using %join%.
- Return statements have the form (RET *expr*). They are treated as the assignment statement (WR (OBJ *return − object − id*) *expr*).
- Call statements have the form (WR *lexpr* (CALL ...)). They are treated as the assignment statement (WR *lexpr* (OBJ *return − object − id*)).
- The *out* mapping equals the *in* mapping for the other cases.

```
/* point is the current node-id */
/* in is the input flow data */

if the intermediate code is a return statement {
    lexpr = return − object − id of the called procedure;
    rexpr = lvalue(get-return-expr(point), in);
    if rexpr is empty
        return in;
    out = cons(dot-product(lexpr, rexpr), in);

} else the intermediate code is a procedure call with assignment {
    lexpr = lvalue(get-left-expr(point), in);
    rexpr = get-obj-mapping(out, return-object-id);
    if lexpr contains one simple object o
        remove all mappings containing o from out;
    out = cons(dot-product(lexpr, rexpr), out);
```

```
} else it is an assignment statement {
    lexpr = lvalue(get-left-expr(point), in);
    rexpr = lvalue(get-right-expr(point), in);
    if rexpr is empty
        return in;
    out = in;
    if lexpr contains one simple object o
        remove all mappings containing o from out;
    out = cons(dot-product(lexpr, rexpr), out);
}
return out;
```

The %call% function defines how parameters are passed and how values are returned. For parameter passing, it computes the lvalues and identifies the object-ids of the formal parameters. A new mapping, *gmap*, is produced by pairing the object-ids of the formals and the lvalues of the corresponding actuals. *gmap* then merges with *in* of the entry node of the called procedure; if the procedure is called from several places, *in* reflects the combination of information flow from each call site. For the return value, %call% collects the *out* mapping from all the exit nodes of the procedure and merges them together using %join%.

```
/* point is the current node-id */
/* in is the input flow data */

pname = get-procedure-name(point);
actual-args = get-args(point);
formal-args = get-formals(pname);
entry-mapping = null;

foreach a in actual − args, f in formal − args do {
    entry − mapping = cons(dot-product(f, lvalue(a, in)), entry-mapping);
    entry − mapping = join(in, entry-mapping);
    exit − mapping = get-exit-vals(pname);
}
return-multiple-values (entry-mapping, exit-mapping);
```

The %diff% function compares two pointer mappings. It first sorts the pointer mappings and then compares them sequentially. Our implementation takes advantage of the monotonic nature of the pointer mapping in flow analysis; the initial mapping is the empty mapping, and each pass over the same node adds new mappings, but never eliminates old mappings. That is, the pointer mapping associated with a node is the same when the number of items in it remains the same.

```
/* $x$ and $y$ are flow data */

$sumx = 0$:
$sumy = 0$;
foreach $mapx$ in $x$, $mapy$ in $y$ do
    $sumx = sumx + length(mapx)$;
    $sumy = sumy + length(mapy)$;
return ($sumx \neq sumy$);
```

The %join% function merges two mappings and removes the duplicates. We first convert mappings into an array indexed by the pointer object. Second, we merge the mappings belonging to the same pointer objects together. Third, we convert the array representation back into the list representation.

```
/* $data - list$ is the list of data to be merged */
/* $result$ is a temporary storage */

$clear - array(result)$;
foreach $mapping - list$ in $data - list$ do
    foreach ($lexpr . rexpr$) in $mapping - list$ do
        $result[lexpr] = set\text{-}union(result[lexpr], rexpr)$;
$out = null$;
foreach $lexpr$ defined in $result$ do
    $out = cons(cons(lvalue, result[lexpr]), out)$;
return $out$;
```

6.5.1.4. Extending APG

The pointer-mapping analysis produces two functions: *pmap* and *lvalue*. *pmap*(*ptr-obj,pointer-mapping*) returns the list of objects that might be pointed to by $ptr - obj$ with flow data *in*. *lvalue*(*expr*, *in*, *point*) returns the pointer value of *expr* at the node *point* with flow data *in*. We also provide $pointer - mapping(node)$ to return the complete pointer mapping associated with *node*.

Both *pmap* and *lvalue* are used to obtain the *gen* and *use* information. *gen* is used to compute reaching definitions, which are described in the next section. *use* is used in the program slicer, described in Chapter 8.

6.5.2. Reaching Definitions

6.5.2.1. Description of Problem

A *definition* of a variable #x# is a statement that assigns, or may assign, a value to #x#. The most common forms of definitions are assignment statements to #x#, in which #x# is definitely modified. They are referred to as *unambiguous* definitions of #x#. Other kinds of statements that may define a value for #x# are called *ambiguous* definitions. The common forms of ambiguous definitions in C programs are the following:

- A side effect of a procedure. The ambiguous definitions include the variables modified in a branch that may or may not be executed.
- An assignment through a pointer. Consider the assignment #*p=y#. The definition is ambiguous if #p# can point to #x# and some other variables. However, the definition is unambiguous if #p# always points to #x#.
- An assignment to a composite object. For example, the assignment #a[2]=y# is an ambiguous definition of #a#.

A definition *d* *reaches* a point *p* if a path exists from the point immediately following *d* to *p* and *d* is not killed along that path. A definition of variable #x# is *killed* if there is an unambiguous definition of #x# along the path. Intuitively, if a definition *d* of some variable #x# reaches point *p*, then *d* might be the place where the value of #x# used at *p* might last have been defined.

6.5.2.2. Preprocessing

The preprocessing involves the identification of the ambiguous and unambiguous definitions and produces bit-vectors representing the definitions generated and killed with each node. In fact, the preprocessing needs more computation time than the flow analysis.

First, we define the *gen* function. A WR-statement or a LIB-statement generates the definitions. The object-ids returned by *lvalue* are the objects defined by a WR-statement. The definition generated with a LIB-statement is predefined, as mentioned in the Data Flow Model described in Section 4.3.4. Under the Data Flow Model, *gen* and *kill* are functions of the actual parameters of the library call because the library call may take some pointers and modify the variables being pointed to. We use pointer mappings to compute the actual list of variables being modified. The following program fragment describes the computation of *gen* and *kill*.

```
/* gen contains both ambiguous and unambiguous definitions */

if node contains a WR-statement
   gen(node) = lvalue(get-left-side(node), pointer-mapping(node));
else if node contains a LIB-statement
   gen(node) = lib-gen(lib-name(node),
               lib-actuals(node), pointer-mapping(node));
else
   gen(node) = null;


/* kill  only contains unambiguous definitions */

if node contains a WR-statement
   if lvalue(get-left-side(node), pointer-mapping(node)) returns { o }
      kill(node) = {o};
   else
      kill(node) = null;
else if node contains a LIB-statement
   kill(node) = lib-kill(lib-name(node),
               lib-actuals(node), pointer-mapping(node));
else
   kill(node) = null;
```

The definitions are uniquely identified by the pair (*object-id*, *node*). We assign a unique integer, $def - id$, to each definition so that we can use the bit-vector representation for the GEN and KILL used in computing reaching definitions. Notice that GEN and KILL are bit-vectors containing definitions, but *gen* and *kill* are functions returning lists of objects. GEN and KILL are defined as follows. In practice, we use a more efficient algorithm for computing KILL.

```
/* M is the number of nodes */
/* D is the number of definitions */
bit-vector  GEN[M][D], KILL[M][D];

initialize GEN and KILL to zeros;
foreach node in the program graph
   foreach var in gen(node)
      GEN[ node ] [ def - id(var, node) ] = 1;
   foreach var in kill(node)
      foreach node2 in the program graph
         KILL [ node ][ def - id(var, node2) ] = 1;
```

We further allocate two bit-vector *in* and *out* for each node and initialize them to the *top* value--all zeros.

6.5.2.3. Solution

The transfer function %code% computes the formula $out = (in - KILL[node]) \cup GEN[node]$. The set-difference function is implemented with the bit-vector *andc*2 function, which bitwise-ands the first argument with the complement of the second argument. The set-union is implemented using the bitwise-inclusive-or function, $bit - ior$. The %code% function is defined as follows.

```
/* point is the current program node */
/* in and out are flow data */

out = bit-andc2(in, KILL[point]);
out = bit-ior(out, GEN[point]);
return out;
```

The $get - proc - gen$ are the formal parameters generated at the entry point of a procedure, and the $get - proc - kill$ remove the formal parameters and the local variables when the reaching definition is passed back to the calling procedure. The reaching definitions at all exit points are collected by the function $get - exit - vals$.

```
/* point is the current program node */
/* in and out are flow data */

pname = get-procedure-name(point);
call-data = bit-ior(in, get-proc-gen(pname));
ret-data = join(get-exit-vals(pname));
ret-data = bit-andc2(ret-data, get-proc-kill(pname));
return-multiple-value (call-data, ret-data);
```

The %join% function is implemented using the bit-wise $bit - ior$ function.

```
/* data - list is the list of flow data to be joined */

result = null
foreach d in data - list
   result = bit-ior(d, result);
return result;
```

The %diff% function is implemented using the block-compare primitive.

6.5.2.4. Extending APG

The preprocessing part introduces the functions *gen*, *kill*, and $def - id$. After flow analysis, we provide the *reach* function. The function *reach*(*obj*, *p*) tells where an object *obj* used at point p is defined.

The reaching definitions, combined with use information, can be used to compute the data-flow dependence among nodes in the program graph. The data-flow dependence will be used in the program slicer.

6.5.3. Node and Predicate Dominance

6.5.3.1. Description of Problem

We say node d of a flow graph *dominates* node n--written d *dom* n--if every path from the initial node of the flow graph to n goes through d. Under this definition, every node dominates itself, the entry of a loop dominates all nodes in the loop, and the testing of an if-then-else statement dominates its two bodies.

Similarly, a predicate p of a flow graph *dominates* node n if every path from the initial node of the flow graph to n goes through p. Notice that the predicate dominance is a syntactic property; semantic equivalence of predicates is not considered.

The following algorithm can be used to determine node dominance.

```
/* $N$ is the set of nodes in the program graph */
/* $n_0$ is the entry node */

while changes to any $dom(n)$ occur do
    for $n$ in $N - lbn_0rb$ do
        $dom(n) = lbnrb \bigcup dom(p)$ where $p$ precedes $n$;
```

The initial approximation of *dom* is that the initial node dominates the initial node, and every other node dominates everything except the initial node. The *top* element is all nodes. Intuitively, the flow analysis rules out the dominator candidates when, for example, it finds a path that proves m *dom* n is false. If we cannot find such a path, from the initial node to n avoiding m, then m dominates n.

6.5.3.2. Preprocessing

In addition to determining the region dominated by a node, we also want to determine the region dominated by the outcome of a branch predicate. We achieve these goals by splitting a branch node n with condition B into two nodes, n_{true} and n_{false}, carrying the predicates B and B, as shown in Fig. 6.6. The nodes dominated by n_{true} are also dominated by predicate B, similarly for n_{false} and B.

Again we use the bit-vector representation for efficiency. Therefore the nodes and the split nodes are assigned a unique number.

6.5.3.3. Solution

With the dominator analysis, we do not distinguish between intra- and inter-procedural flow. Instead of implementing another algorithm, we take advantage of the developed flow algorithm to compute the dominance problems.

The %code% function is simply defined as:

```
/* node is the current node */

out = in;
out[node] = 1;  /* a node dominates the nodes following it */
```

The %call% function is the null function because the parameter passing and return values do not affect the analysis.

The %diff% function is set-unequal, which is implemented using the bit-vector comparison function.

The %join% function is set-intersection, which is implemented using the bitwise-inclusive-or function.

6.5.3.4. Extending APG

The *dom*(*node*) function determines the nodes that dominate *node*. The application finds out the coverage of the predicates in the branch statements. The coverage is useful in the program slicer (Section 7.3). *pdom*(*branch-node*, *true*) further distinguishes the coverage of the predicate when the branch is taken, and *pdom*(*branch-node*, *false*) does this when the branch is not taken. *pdom*() is useful in the generation of verification conditions (Section 8.6) by identifying the valid predicates.

6.5.4. Acyclic Reaching Definitions

When a variable is defined, it *depends* on the variables used on the left side of the assignment statement. In other words, when a variable is defined in a loop, it may depend on its definitions of previous iterations. For example, a variable that may depend on its own value is an *induction* variable. This creates cycles when we use the reaching-definition analysis to generate a data dependence graph. The cyclic data dependence is useless and undesirable (when used in the well-behavedness checker), so we develop this technique to remove the cycles from the dependence graph.

To implement acyclic reaching definitions, we introduce unambiguous definitions at the loop entry point for every induction variable in each loop. (The detection of induction variables is described in Section 8.4.2.) The additional definitions kill the reaching definitions that are both generated inside the loop and propagated back to the loop entry. Therefore, the dependence cycles are broken.

We use the reaching-definition mechanism to compute acyclic-reaching-definition analysis. The only change is in the initialization. Every induction variable at the loop entry is added to *gen*, and all other definitions of the variable are added to *kill*.

Since acyclic reaching definitions will not propagate around loops, two passes are sufficient for the flow-analytic algorithm to terminate for non-recursive programs.

Similar to *reach*(*obj*, *p*), the function *areach*(*obj*, *p*) tells in which assignment statement or in which loop invariant the object *obj* used at point *p* is defined. The acyclic reaching definition is used in the well-behavedness checker to generate verification conditions (see Section 8.6). The generation of loop invariants (see Section 8.4) recovers the information lost through the elimination of the cyclic dependence.

6.5.5. Integer Ranges

6.5.5.1. Description of Problem

An integer range is expressed with two integers (*min*, *max*), which denote the minimum and maximum values of an integer variable. The integer-range analysis determines the range of the array-subscript expressions in the well-behavedness checker.

An integer range is generated when a constant is assigned to a variable. It is computed when variables are added, subtracted, multiplied, and divided. It is propagated along all program paths. When two paths merge, the integer ranges are combined. When integer ranges reach a branch node, the branch predicate refines the ranges. Furthermore, if the program is annotated with assertions or loop invariants, they can also be used to refine the ranges.

Compared with other flow problems, the integer-range analysis examines programs that have already been enhanced with extra information--the loop invariants discovered by the well-behavedness checker. Another difference is that integer-range analysis needs many supporting functions such as range arithmetic, union, intersect, and range refinement.

6.5.5.2. Preprocessing

In the integer-range analysis, the data propagated is the mapping $irange: var \rightarrow (min, max)$. Initially, all variables have the *bottom*[2] value--the range $(-inf, \infty)$.

The supporting functions are defined as follows: n is a program node, and *irange* is the input value; *RangeUnion* is used to combine ranges when paths merge; *RangeRefine* is used to refine ranges with the branch predicates; *RangeExpr* and *RangeVar* evaluate the ranges of integer-expressions. The details of their definitions follow:

- $RangeUnion(f, g) = hwhereh(v) = (min(f(v), g(v)), max(f(v), g(v)))$
- $RangeIntersect(f, g) = hwhereh(v) = (max(f(v), g(v)), min(f(v), g(v)))$
- $RangeExpr(E, irange)$ computes the ranges of the expression E given the current integer-range *irange*. It can compute +, -, *, and /.
- $RangeRefine(B, irange)$ uses the linear restraint B to generate tighter ranges for *irange*. B is a normalized boolean expression consisting of AND, OR, and linear inequalities. $RangeRefine(B, irange)$ is defined as:
  Case 1: B is of the form $And(B_1, B_2, \ldots)$
  $irange$ = the fixed-point of
  $RangeIntersect(RangeRefine(B_1, irange), RangeRefine(B_2, irange), \ldots)$
  Case 2: B is of the form $Or(B_{1,}B_2, \ldots)$
  $RangeIntersect(irange,$
  $RangeUnion(RangeRefine(B_1, irange), RangeRefine(B_2, irange), \ldots))$
  Case 3: B is a linear inequality of n variables: $X_1, X_2, \cdots, X_n$

---

[2] All previous flow analysis uses the *top* value. Section 6.6.1 explains the details.

a) more than two variables have *bottom* range

$irange$.

b) $n-1$ variables are defined. (i.e., X_i is *bottom*)

$RangeIntersect(irange, RangeVar(X_i, B, irange))$.

c) all variables are defined.

$RangeIntersect(irange, RangeVar(X_1, B, irange), RangeVar(X_2, B, irange), \ldots)$

- $RangeVar(X, B, irange)$ computes the possible values for X given $irange$ and the inequality B consisting of $+, -, *, /, <, >, =, \leq, \geq$. For example, $RangeVar(K, I \leq J + K, irange)$ generates the bound $(4, \infty)$ for K when $irange(I) = (10, 20)$ and $irange(J) = (-5, 6)$ (i.e., K is 4 when I is 10 and J is 6). $RangeRefine(B, irange)$ refines the possible ranges when B is true. For example, if a path with condition #I<=J+K# is taken, given $irange(I) = (10, 20)$, $irange(J) = (-5, 6)$, and $irange(K) = (0, 9)$, then the refined ranges are $irange(I) = (10, 15)$, $irange(J) = (1, 6)$, and $irange(K) = (4, 9)$.

6.5.5.3. Solution

When this analysis is performed, the well-behavedness checker has generated loop invariants and assertions for the program. We can first take advantage of the loop invariants, $Assert_n$, to refine the integer ranges. Second, if the node contains an assignment statement, the integer range for the right-hand side expression is computed, and a new range is generated for the assigned variable. If the node is a branch node, the branch condition refines the ranges.

```
/* $n$ is the current node */
/* $in$ is the flow data input */

$irange' = RangeRefine(Assert_n, irange)$;
if $n$ is an assignment node of the format $X := E$
   return $irange'[X \rightarrow RangeExpr(E, irange')]$;
else if $n$ is an branch node with condition $B$
   return $RangeRefine(B, irange')$;
else
   return $irange'$;
```

The %call% function computes the ranges for the actual parameters in a procedure call, and generates the new ranges for the formal parameters. If the procedure returns an integer, the range of the return expression is passed back to calling node.

The %diff% function is the list-difference.

The %join% function is *RangeUnion*.

6.5.5.4. Extending APG

The *irange*(*expr*, *node*) is provided to APG to return the range of the integer expression *expr* at *node*.

6.6. Discussion

We conclude this chapter by discussing a few details involved in implementing the flow analyzer. They include the initialization requirement for termination of the flow analysis, the selection of data structure for flow data, the experience of implementing the flow analyzer, and the possibilities of using more complicated and faster algorithms.

6.6.1. Flow Data Initialization

We usually use the *top* value for the initial value of flow data. The *top* and *bottom* element have the following properties:

join(*top*, *x*) = *x*;
join(*bottom*, *x*) = *bottom*;

In many flow problems, we use two types of %join% functions: intersection and union. We use the intersection function when we compute "must" information--information coming from all predecessors. We use the union function when we compute "may" information--information coming from at least one of the predecessors.

Obviously, if %join% is union, then *top* must be the empty set. Similarly, if %join% is intersection, then *top* must be the universal set. For example, the reaching-definition problem has union (bitwise-inclusive-or) as %join% and the empty set (bit-vector with all zeros) as *top*; the available-expression problem has intersection as %join% and all available expressions as *top*.

The initialization of a flow problem is simple if we can just use the *top* element. In fact, it is the case for the pointer-mapping, reaching-definitions, and acyclic-reaching-definitions analyses. A minor problem occurs with available expressions. Using *top* as initial values assumes that all expressions are available on the entrance to a node *n*; however, there might be paths from the initial node to *n* along which an expression is neither generated or killed. Initializing the initial node with *bottom*--the empty set--fixes this problem.

The node dominance uses intersection as %join% and the universal set as *top*. If we use *top* as the initial value, then every node dominates every other node. Initializing the entry node with *bottom* fixes this problem.

The initialization for integer-range analysis is most troublesome. %join% is the union of integer-ranges. Therefore, the *top* value is $(\infty, -\mathit{inf})$. Consider the following example:

```
1  i = 1;
2  while (i < 10000)
3     i++;
```

If we initialize the range of #i# to $(\infty, -\mathit{inf})$, then the range before statement 3 is (1,1) at the first pass, (1,2) at the second pass, (1,9999) at the 9999th pass, and stabilizes thereafter. The worst time for the flow analyzer to converge is during counting of the minimum integer to the maximum integer. This problem does not happen with other analyses such as the reaching-definition and dominator because their flow data are boolean, and one pass through the node is sufficient to determine a boolean property.

If we initialize the range of #i# to $(-\mathit{inf}, \infty)$, then the range before statement 3 is $(1,\infty)$ at the first pass, and stabilizes thereafter. Since the flow algorithm stops when the flow data does not change, the results are always invariants of the program. However, the result $(1, \infty)$ produced with *bottom* initial value is not as precise as the desired result (1,9999) produced with *top* initial value. In Section 8.5, we present several techniques (e.g., loop invariant generation, def-use information, identification of interval-constant) to improve this situation.

6.6.2. Data Structure for Flow Data

From our limited experience, there are two rules of thumb in solving flow problems in our framework.

- **Use a bit-vector representation as much as possible.** In the first prototype, we used a list representation for the reaching definition to save the preprocessing. By reimplementing it with the bit-vector implementation, the speedup is greater than one hundred times.
- **Preprocess everything that you can.** The preprocessing takes place only once, but the flow analysis needs several passes. By converting the flow information into a data structure more convenient for flow analysis, the computation is greatly improved. For example, we encode the transfer functions using bit-vectors in computing the reaching

definitions, node and predicate dominators, and acyclic reaching definitions.

6.6.3. Building Flow Analysis Tools

We built a general flow-analysis tool to solve the flow problems required for the program slicer and well-behavedness checker. We implemented the framework with 178 lines of Lisp code. However, we used 979 lines of code to develop the required APG functions, such as *pred*, *succ*, *formals*, etc. This number does not include the lines of the code that translates C programs into the APG representation. Figure 6.7 lists the length of the Lisp subroutines used to implement various flow problems. The left column identifies the problem being solved, the middle column shows the size of code to implement the flow functions, and the right column shows the amount of code to implement the supporting functions. For example, we used 469 lines to implement the integer-range arithmetic, union, and intersection functions.

Flow Problem	Solutions	Supporting Functions
Pointer-Mapping	186	0
Reaching-Definitions	215	169
Node and Predicate Dominators	119	0
Acyclic-Reaching-Definitions	242	0
Integer-Range	223	469

Although the framework does not significantly reduce the programming effort, it does provide a good foundation to define and handle new flow problems. Sharing a common framework, the results of various flow analyses can be easily integrated together.

Furthermore, the work on the framework is much less than the work to generate the intermediate form or the work to define the problem. The most important feature of a flow-analysis tool is the intermediate form, which defines the available information and the properties that the flow analyzer can assume.

6.6.4. Faster Algorithms

Faster algorithms can solve the flow problems discussed above by identifying the specific properties of each problem. The faster algorithms provide ways to improve the execution time of some of the flow analyses. Unfortunately, the pointer-mapping and integer-range analyses remain the bottleneck of the system, no matter how fast the other problems can be solved. The following briefly discusses how to improve flow problems discussed.

- Cooper [Cooper,fast,alias] proposed an algorithm with $O(n^2 + ne)$ steps to compute equivalence aliasing at the procedure level. His algorithm divides intra- and inter-procedural analyses into two phases. He took advantage of the fact that the equivalence aliasing transfer functions are closed under composition. A set of functions, F, closed under *composition* means that if f and g are functions in F, $f \circ g$ is also a function in F. The pointer-mapping transfer functions are not closed under composition, therefore it cannot be solved with Cooper's approach.
- Using interval approach, the reaching definitions can be solved in $O(n \log n)$ time. However, normal C programs may not satisfy the reducible-graph property. Node splitting [Ullman,compiler] is required to convert a program graph into a reducible graph.
- The acyclic reaching definitions already have an $O(n)$ solution, so there is no need to improve it.
- Lengauer and Tarjan [Lengauer,dominators] presented a fast algorithm to determine dominators. Their algorithm runs in $O(e\alpha(e,n))$ time where $\alpha(e,n)$ is a functional inverse of Ackermann's function. [introduction,algorithms]
- The integer-range analysis cannot be expressed using composable transfer functions. The interval approach requires the composable property. Therefore, we cannot use interval approach to solve the integer-range problem.
- The summary-graph approach [Chow,analyzer] can speed up pointer-mapping and integer-range analyses. The summary graph approach is applied to pointer-mapping analysis by first eliminating the nodes that have no pointer assignments. Then analysis is performed on the reduced graph. This method does not improve the asymptotic complexity, but can reduce the execution time by an order of magnitude by effectively reducing the program size.

To build a faster flow analyzer, we should use the summary-graph approach to compute pointer mappings and integer ranges, which consume the most time consuming in the analysis.

CHAPTER 7

Program Slicer

7. _7

Program slicing [weiser,slicing] produces a bona-fide program--a subset of the original program that behaves exactly the same with respect to the computation of a designated property. Section 7.1 first discusses Weiser's idea on program slicing. [weiser,slicing] Section 7.2 describes how to compute the control-dependence and data-dependence information required to generate a program slice. Weiser's slicing algorithm is, in fact, only one of many ways to slice a program. Section 7.3 investigates different ways to slice a program. Section 7.4 briefly introduces the importance of selecting the slicing property. Chapter 8 identifies the appropriate slicing properties for malicious-code detection.

7.1. Overview

Program *slicing* is a method used by experienced programmers to abstract from programs. [Weiser,program,slicing] Starting from a subset of a program's behavior, slicing reduces that program to a minimal form that still produces that behavior. Automatic slicing requires that behavior of interest be specified in a certain form. If the behavior of interest can be expressed as values of some set of variables at some set of statements, then this specification is said to be a *slicing* *criterion*. [weiser,slicing]

Slicing is useful in debugging, testing, and maintenance of software when only a portion of a program's behavior is of interest and must be examined. Program slicing identifies the portion of the program that affects the specified behavior and produces slices that are significantly smaller and simpler than the original program.

Finding the exact (i.e., smallest) program slice (in the semantic sense) is unsolvable. However, data-flow analysis can determine an approximate program slice by finding all of the program code that might (instead of actually) have influenced the specified behavior. One possible way to slice a program is to delete source statements, but the sliced source program may then become ungrammatical. For example, removing the THEN clause from an IF-THEN-ELSE statement leaves an ungrammatical fragment if the “null” statement is not permitted between THEN and ELSE. A better but more complicated way is to translate the program into an intermediate form, slice the intermediate program, and then reconstruct the slice into a complete program.

Figure 7.1 illustrates program slicing. It contains a statistics program that computes the mean, the variance, and the product of the input data. We want to slice the part of program that computes the mean. Before computing the program slice, we first process the program through the translator and flow analyzer. The translator decomposes compound statements into simple intermediate statements (e.g., line 09 decomposes into 6 intermediate statements) and assigns unique object-id for memory objects. The flow analyzer computes the pointer mappings and reaching definitions. The program slicer computes the control and data dependence, which are described in detail in the next section.

```
00  int data[MAX]; 01  statistic(n) 02  int n; 03  { 04     int i, sum, sum_square,
*iptr; 05     float prod, mean, var; 06     sum  = 0; 07     sum_square = 0; 08
prod = 1.0; 09     for (i = 0, iptr = data; i < n; i++, iptr++) { 10        sum  +=
*iptr; 11        sum_square += *iptr * *iptr; 12        prod *= *iptr; 13     } 14
mean = (float) sum / n; 15     var  = (float ) (sum_square - sum * sum) / n; 16
printf("mean: %f\n", mean); 17     printf("var: %f\n", var); 18     printf("prod:
%f\n", prod); 19  }
```

```
01  statistic(n) 06     sum  = 0; 09     for (i = 0, iptr = data; i < n; i++, iptr++) { 10
sum  += *iptr; 13     } 14     mean = (float) sum / n;
```

The dependence information relates one statement to the others. For example, at line 14, we need the values of #sum# and #n# to compute #mean#. Data-flow analysis tells us which definitions of #sum# and #n# are used, i.e., #sum#:10, #sum#:6, #n#:1. (We use the notation Var:Line to indicate the Var's value defined at line Lineno.) #sum#:10 is defined by #sum#:10, #sum#:6, and #iptr#:9. Collecting the statements generating these definitions, the program slice expressed in C fragments is in Fig. 7.2. Technically, the slice should also contain the variable declarations and the brackets around the body, but our slices do not.

7.2. Overlaying Control and Data Dependence with APG

APG is not sufficient to be directly used for program slicing. We build layers of abstraction on top of APG: data-flow graph and predicate-region, based on Ferrante's result. [Ferrante,predicate,dependency] We also compute the precedence relations for APG nodes. Based on this dependence information, we can build the program slicer easily.

7.2.1. Data-Flow Graph

The *data* *flow* *graph* is constructed by linking every use (variable reference) to all reaching-definitions, which makes definition-use chaining (du-chaining) explicit for each APG node. *Du-chaining* identifies for a point p the set of uses s of a variable, say #x#, such that there is a path from p to s that does not redefine #x#.

For each APG node n, we identify its definitions (variable modification) and uses (variable references). Definitions are produced in only two situations: explicit modification using the WR operator, and implicit modification using the LIB operator. For the former, we obtain the first argument of WR and apply *lvalue* to determine which objects are modified. For the latter, we call the APG function $lib - def$ to find out which objects are modified. The computation of use is more complicated, including the variables used to generate the lvalue on the left side, and also the variables used to generate the rvalue on the right side. We collect the expressions and then recursively determine which variables are used. For example, #b# and #c# are used in the expression #a[b]=c#, and #q#, #p#, #lvalue(p)#, and #lvalue(lvalue(p))# are used in `*q =**p;`.

The *use* information is then stored back to APG. We then use the reaching information to generate the data dependence. $reach(obj, n)$ returns the definitions of object obj reaching node n; each definition is represented as an ordered pair $(object, node)$. Two functions are added to APG for tracing through the data-dependence graph forward and backward. $forward - depend(obj, n)$ identifies the variables computed using the value of the object obj defined at node n. $backward - depend(obj, n)$ identifies the definitions affecting the definition of obj at node n.

```
foreach node $v$ in the program graph
  foreach $obj - def$ in $def(v)$
    foreach $obj - use$ in $use(v)$
      foreach $(obj, w)$ in $reach(obj - use, v)$
        add $w$ to $backward - depend(v)$
        add $v$ to $forward - depend(w)$
```

Similarly, we define the closure versions of $forward-depend$ and $backward-depend$. The parameter *focus* is used to narrow down the scope of the program we are examining.

```
$forward-depend * (node-list, focus)$ ==
  foreach $node$ in $node-list$ when $node$ in $focus$
    $node-list = node\text{-}list \cup forward\text{-}depend(node)$
  return $node\text{-}list$ intersection $focus$;

$backward-depend * (node-list, focus)$ ==
  foreach $node$ in $node-list$ when $node$ in $focus$
    $node-list = node\text{-}list \cup backward\text{-}depend(node)$
  return $node\text{-}list$ intersection $focus$;
```

7.2.2. Predicate Region

Each control structure defines a region of influence (*predicate* *region*) [Ferrante,1983] that may be labeled with the branch predicate. We build a control-dependence graph linking the predicate and the region of its influence. We further partition Ferrante's predicate region into the true sub-region and false sub-region.

For structured programs, a predicate region can be easily constructed using the node dominance result from the flow analyzer. An APG node n is in the predicate region of branch node b if b dominates n. Intuitively, if there is no jump into an enclosed block, the predicate must be in every path reaching any node in the control structure.

For non-structured programs, this simple approach lacks certain control-dependence information. Consider the example in Fig. 7.3, which cannot be expressed structuredly without duplicating code. Predicates A_{true} and B_{true} influence node C, but neither of them dominates C.

A better solution by Denning and Denning [Denning,secure,information,flow] uses the nearest inverse dominator of a branch statement to define its range of influence. An inverse dominator of node n is on every path from n to the exit point of the program graph. Intuitively, the nearest inverse dominator is the node that can be reached from either side of a branch node; therefore, the range of influence of a predicate terminates after such a node. The predicate region is where the nodes can be reached from a predicate p without passing through the nearest inverse dominator of p. Consider the example in Fig. 7.3 again. D is the nearest inverse dominator for node A and node B. The predicate region for B_{true} is {C}, for B_{false} is { }, for A_{true} is {C}, and for A_{false} is {B}.

The implementation of the more accurate predicate region requires using backward-flow analysis that has not been supported by the flow analyzer. In our prototype, we have implemented the simpler, but less accurate predicate-region analysis based on dominators.

Two functions, which are inverses of each other, are added to APG: $predicate-depend(x)$ and $predicate-region(y)$. They are defined as follows:

```
foreach node $n$
  foreach $d$ in $dom(n)$ such that $is-branch(d)$
    add $d$ to $predicate-depend(n)$;
    add $n$ to $predicate-region(d)$;
```

Similarly, we determine other versions of $predicate-depend$ and $predicate-region$ that take a list of definitions and a focus as inputs.

```
$predicate-depend*(node-list, focus)$ ==
  foreach $node$ in $node-list$ when $node$ in $focus$
    $node-list = node\text{-}list \bigcup predicate\text{-}depend(node)$;
  return $node\text{-}list\ intersection\ focus$;

$predicate-region*(node-list, focus)$ ==
  foreach $node$ in $node-list$ when $node$ in $focus$
    $node-list = node\text{-}list \bigcup predicate\text{-}region(node)$;
  return $node\text{-}list\ intersection\ focus$;
```

7.2.3. Node Precedence

Node dominance computes the “must” information--a dominator of v is a node that appears on every path from the entry point to v. Node precedence computes the “may” information--determining the nodes that appear on some paths from the entry point to v. The node-precedence relation is needed for program slicing. We apply a transitive-closure algorithm using APG functions *pred* and *succ* to compute the node precedence. Two new functions, $pred*(n)$ and $succ*(n)$, are generated. $pred*(n)$ returns the list of nodes that may precede n, and $succ*(n)$ returns the list of nodes that may succeed n.

7.3. Program Slicing

The concept of breaking down a large program into smaller modules for analysis dates back to 1975. [Zislis,semantic,decomposition] Zislis uses busy variables (variables that will be used later in the program) as the criteria to group related program statements together and form a slice. Weiser uses a more accurate criteria, data dependence, to group statements together. These criteria are not the only ways of grouping relevant and eliminating irrelevant statements.

In this section, we discuss several ways of applying the control-dependence and data-dependence analyses to "slice" a program--namely, backward data-flow slicing (Weiser-style slicing), forward data-flow slicing, predicate-region slicing, and control-flow slicing. These ways are applied for different purposes but share the same platform for analysis.

A program slice is represented by a set of nodes. Given the set of nodes and APG, a subset program can be reconstructed easily. Since a program slice is represented by a set, it is possible to combine the effect of different slicing methods by set-union, set-intersection, and re-slicing using different criteria. In the following discussion, *focus* is used to combine different slicing methods. Notice that *focus* initially contains the whole program.

7.3.1. Control-Flow Slicing

Control-flow slicing is extremely simple. Since there is no reason to look at complete execution paths all the time, we can eliminate those sections in which we are not interested. For example, when slicing for the file accesses call, we are interested in sections of paths starting at the entry point and ending at an #open()# system call. For programs including authentications, we may only be interested in the authentication section.

The control-flow slicer accepts two points--u and v--in a program and determines the nodes in any path going from u to v. The slicing is produced by the following equation:

$$focus = focus\ intersect\ succ^*(u)\ intersect\ pred^*(v)$$

7.3.2. Backward Data-Flow Slicing

Weiser uses backward data-flow slicing. Informally, it determines which statements affect the variables at the statement under examination. A statement can affect a subsequent statement either directly or indirectly. The *direct* effect provides a value to be used at the later statement. The *indirect* effect controls whether the later statement will be executed. In the following example, statement 3 has a direct effect on 4 because #y#:4 uses the value #x#:3; statement 2 has an indirect effect on statements 3, 4, and 6 because of the branches.

```
1  c = 1;
2  if (c) {
3      x = 10;
4      y = x;
5  } else
6      y = 3;
```

The slicing algorithm, shown in Fig. 7.4, is a loop collecting the direct and indirect effects. The algorithm accepts a node as the slicing point. The variable *focus* usually carries the part of the program narrowed down by control-flow slicing.

7.3.3. Forward Data-Flow Slicing

Forward data-flow slicing determines the effect of certain computations in the program. It is very similar to backward data-flow slicing, except that it traces forwards through data-flow graph and predicate-regions. The algorithm is shown in Fig. 7.5.

Input: *node*
Output: the nodes that constitute the program slice.

```
/* Initialization */
$new - list = \{node\};$
$node - list = \{\};$

/* repeat until we cannot collect more nodes */
while ($node\text{-}list \neq new\text{-}list$) {
    $node\text{-}list = new\text{-}list;$
    /* Collect Data Dependence */
    $new - list = backward - depend * (node - list, focus);$
    /* Collect Control Dependence */
    $new - list = predicate - depend * (new - list, focus);$
}

output $node - list;$
```

Input: *node*
Output: the nodes that constitute the program slice.

```
/* Initialization */
$new - list = \{node\};$
$node - list = \{\};$

/* repeat until we cannot collect more nodes */
while ($node\text{-}list \neq new\text{-}list$) {
   $node\text{-}list = new\text{-}list;$
   /* Collect Data Dependence */
   $new - list = forward - depend * (node - list, focus);$
   /* Collect Control Dependence */
   $new - list = predicate - region * (new - list, focus);$

}

output $node - list;$
```

7.3.4. Other Slicings

Both the forward and backward data-flow slicing algorithms combine the collection of direct and indirect effects. The time-bomb example in Section 2.5.2 requires a different kind of program slicing, in which the direct effects are collected first, and then indirect effects are identified. The time-bomb slicing algorithm can be built as follows:

```
$new - list = \{node\};$
$node - list = \{\};$

/* Collect Data Dependence */
/* Repeat until we cannot collect more nodes */
while ($node\text{-}list \neq new\text{-}list$) {
   $node\text{-}list = new\text{-}list;$
   /* Collect Data Dependence */
   $new - list = forward - depend * (node - list, focus);$
}

/* Collect Control Dependence */
$node - list = predicate - region * (node - list, focus).$

output $node - list;$
```

Another possible slicing option is to ignore the control-dependence analysis. When examining programs for security-related properties, it is usually sufficient to examine the worst situation. Consider analyzing the who program in Fig. 7.6. As the first approximation, we just want to know which files are accessed, but do not need to identify under exactly which conditions the files are used. The backward data-flow slice, given in Fig. 7.7a, is a 4-line program. The approximate slice, given in Fig. 7.7b, ignores the control information and is a 3-line program obtained from 7.7a by dropping the if-statement. In the analysis of file accesses, we have to evaluate the program, hence the content of s, under all possible inputs. The extra information produced by the condition #argc==2# does not improve the analysis; therefore it can be dropped.

If the file access part is in a loop, the #open()# system calls are control-dependent on the loop predicates. The normal program slicing will include the loop predicates and all the statements affecting the variables used in the predicate. Ignoring the control-dependence in this case will result in a much smaller program slice.

```
main(argc,argv); char *argv[]; {              ....          /* many lines deleted */
     s = "/etc/utmp";
     if(argc == 2)
          s = argv[1];              ....          /* many lines deleted */
     if ((fi = fopen(s, "r")) == NULL) {
          puts("who: cannot open utmp");
          exit(1);
     }
     while (fread((char *)&utmp, sizeof(utmp), 1, fi) == 1) {
          if (argc == 3) {              ....          /* many lines deleted */
```

```
s = "/etc/utmp"; if (argc == 2)
   s = argv[1]; fopen(s, "r") ...
```

7.7a. Backward data-flow slicing

```
s = "/etc/utmp";
   s = argv[1]; fopen(s, "r") ..
```

7.7b. Approximate slicing

7.4. Discussion

We built a program slicer for APG programs. We implemented several similar program-slicing methods, and used the program slicer to examine various tell-tale properties.

Weiser [weiser,debugging] noted that many slices using different property usually end up being very similar. In his experiment with student-written compilers, the number of distinct slices (i.e., different by more than ten statements) was relatively small, and the size of program slices ranged from less than 1% and up to 96% of the program, averaging 44%.

Our experience is similar to Weiser's. We noticed that the size of a program slice depends on the slicing property, and very often several different slicing properties result in the same slice. The effort required to analyze a program depends on the number and the sizes of the program slices. Chapter 8 discusses the selection of properties indicating that a small slice will be produced. In some cases, distinct properties can be determined using a common slice.

Unfortunately, the primitive program slicer generates a set of APG nodes constituting the program slice, which is very difficult to read. We believe a pretty printer program that converts sliced APG programs back into C programs is useful to the human analyst. The conversion poses no technical difficulties, although it does require a fair amount of programming: generating C statement from APG nodes, generating high-level constructs from the flow graph, [Baker,structuring] and collecting relevant variable and type declarations.

CHAPTER 8

Well-Behavedness Checker

8. _8

A program that has neither out-of-bounds array accesses nor overflowed pointer dereferences is well-behaved. Since our program slicer can be fooled by malicious programs that are not well-behaved (see Section 3.2.2), we build a well-behavedness checker to detect them. Although a benign program may not be well-behaved, this lack of well-behavedness typically is due to programming error.

We validate well-behavedness statically and automatically using program-verification and flow-analysis techniques. Different from traditional approaches, our approach is extensible and runs with almost linear complexity in practice. For generality, we use extensible template matching to identify the loop invariants. With only seven templates, we can recognize many common array-indexing situations. The loop invariants discovered are used to strengthen the integer-range analysis, which is used to determine the bounds of the array-subscript expressions. For robustness, those array checks not verified by flow analysis are handled by verification techniques. Verification can be used to analyze array accesses directly, but it is expensive. We first use flow analysis to minimize the number of checks necessary before using verification techniques. The verification-condition generator produces relatively simple theorems to be proved by identifying relevant information of the array bound check using data-flow information. It is also less computationally intensive than traditional verification-condition generators.

Section 8.1 reviews the well-behavedness property. Section 8.2 presents the previous work on array bounds checking. Section 8.3 discusses our integrated approach. Section 8.4 introduces the generation of loop invariants using templates. Section 8.5 describes how the result of integer-range analysis is used to validate array accesses. Section 8.6 shows how a verification condition is generated in the case when array accesses fail to be validated. Section 8.7 discusses the preliminary implementation. Section 8.8 discusses the problems and limitations of automating this process.

8.1. The Well-Behavedness Property

The well-behavedness property guarantees that a program satisfies our data model, alias model, and data flow model in the semantic sense. It can be summarized as follows:

- Array accesses are not out-of-bounds. For example, if #a# is an array with 10 elements, #a[10]# and #a[-1]# are out-of-bounds array accesses.
- Pointers are well-formed lvalues. An lvalue is well-formed only if it is composed of an address of a variable and an integer offset. For example, #&a#, #&a+10# are well-formed lvalues, and #&a+&b#, #120#, #&a*10# are not.
- Pointers are not overflowed. Pointer overflowing is equivalent to the out-of-bounds array accesses problem. When the pointer expression is a well-formed lvalue and the address part is an array variable, the offset must be within the bounds of the array variable; if the address part is a simple variable, the offset must be zero; if the address part is a structure, the offset must be one of the fields of the structure.

Checking that pointers are well-formed is performed in the pointer-mapping analysis, in which the mapping from pointers to variables is determined. The pointer-overflow problem is translated into an array-overflow problem. The well-behavedness checker attempts to verify that no out-of-bounds array accesses exist.

8.2. Background on Array-Bound Checking

To aid in the debugging of programs under development, many compilers generate run-time checks to detect array bound violations. However, run-time checking may unacceptably lengthen the program execution time. In cases where we cannot handle run-time errors, we must statically demonstrate the correctness of the array accesses. To verify a program against its specification, we must prove the correctness of all array accesses in the program. [Boyer,1981] Software quality assurance and security software analysis also require this verification.

Previous approaches to this problem can be divided into three main categories: program verification [Suzuki,1977], mechanical flow analysis, [Harrison,1977] and array-check optimization. [Gupta,1990]

In the program-verification approach, the correctness of array accesses is stated in the framework of weak correctnesses [Suzuki,1977] of programs. That is, we only verify the assertions related to the subscript expressions in array accesses. Typically, a verification-condition generator (VCG) first converts a program proof (together with the input and output assertions) into predicates [Floyd,1967] called verification conditions (VC). Then the VCs are proved. The general program-verification approach can handle very complicated programs, but the programmer must provide the appropriate loop invariants, which are predicates characterizing the behavior of a loop during program execution. The major difficulties in automating program verification are the automatic generation of loop invariants [German,1975] [Wegbreit,1973] and the proof of verification conditions. [Boyer,1988]

When mechanical flow analysis is used, programs are analyzed to determine bounds on the ranges of values assumed by variables at various points in the program. Flow analysis [Kennedy,1981] is used to propagate and refine the range information. Integer-range analysis [Harrison,1977] complements range propagation by deriving simple inductive definitions of the value ranges of variables. For efficiency reasons, only monotonic recurrence relations able to be described in the format $Var_{i+1} = Constant_1 * Var_i + Constant_2$ are used. Cousot [Cousot,linear] also uses flow analysis to solve this problem and describes relations using linear constraints among variables. His approach is based on the propagation of systems of linear inequalities. During the propagation, the convex-hull of a system of linear inequality is computed and the system simplified to a minimal one. This approach discovers more relations than Harrison's method, but requires a lot of computation. For the convergence of the analysis, Cousot uses a heuristic (widening operator) to discard the restraints that do not quickly stabilize during the analysis. Using heuristics allows a compromise between the convergence and the loss of information.

The array-check optimization [Gupta,1990] employs optimization techniques such as elimination of redundant checks (when one check implies the other) and code motion (moving checks out of loops) to reduce the number of necessary checks. When used in the run-time array bounds checking, it reduces the execution overhead involved. When used in static approaches, it reduces the number of assertions to be verified.

8.3. Overview of the Well-Behavedness Checker

We integrate flow analysis, optimization, and program-verification techniques to analyze array bounds statically. Our approach covers many situations and is less computationally demanding than previous approaches. We produce stronger loop invariants, express information in propositional logic and systems of linear inequalities, and process those assertions using the Boyer-Moore [Boyer,1988] theorem prover. For improving efficiency, we use flow analysis to handle the most common and simple array access situations, eliminate redundant checks as a side effect of the flow analysis, and generate one theorem to be proved for each remaining array access. This approach has three major steps: invariant generation, integer-range analysis, and VC generation, as shown in Fig. 8.1. The translator and flow analyzer provide the preprocessing.

The first step generates loop invariants. Generating the correct loop invariants is difficult, and generally requires several iterations. The deficiency of both program verification and mechanical flow analysis contributes to the lack of loop invariants. Suzuki [Suzuki,1977] uses the *Induction-Iteration* *Method* to generate loop invariants automatically by propagating array-access predicates through loops and combining them with various branch conditions. This method generates weak invariants, which are usually strong enough to verify array bounds but too weak to verify program correctness. The disadvantages are that it may never converge, it uses theorem proving to decide the convergence, and the size of the assertion grows exponentially with the number of iterations. In the integer-range analysis method, [Harrison,1977] only monotonic recurrence is expressed and very simple loop invariants are recognized. We use program templates to recognize loop invariants. This matching, which is localized to the induction variables in a loop, is much faster than propagating assertions, especially when the program is large. The loop invariants we generate, in the form of linear inequalities, are usually powerful enough to express the relations between most integer expressions used as array subscripts in the program. The system can make use of the array-check assertions tentatively to strengthen the loop invariants. Of course, the tentative assertions must then be justified. When the automatically generated invariant is not powerful enough, the system can accept the programmer's assertions to strengthen the invariants.

The second step uses flow-analysis techniques: integer-range propagation and refinement. Like program verification, we use the loop invariants generated and the tentative assertions (array-bound-check predicates or the supplied assertions) to restrict the value ranges. After the flow analysis terminates, the tentative assertions are checked, and eliminated if they are proved valid. Similarly to an inductive proof in program verification, a tentative assertion may justify itself through the flow analysis. A portion of the array access checks will then be validated, another portion will be eliminated simply because it is covered by the others, and the remaining portion will be handled using verification techniques.

The third step generates the verification condition for the array bounds checking and uses a theorem prover to prove those assertions. The verification-condition generation (VCG) transforms the array-access problem from the program verification domain into the theorem proving domain. That is, the array-access property is verified only if the generated assertions are proven correct. In a traditional VCG, the verification proof of one assertion usually induces many to be proven because of the propagation of the assertion throughout the loops and branches in the program. We employ the loop invariant generated in the template matching to flatten the program into an acyclic form. While renaming the induction variables at each loop header (cut-point) and variables defined at each definition point, we compute a new Def-Use chain that is acyclic. For each array-check predicate, we trace through the Def-Use chain to generate a verification condition. The verification condition so generated is expressed in propositional logic and systems of linear inequalities. Proving the theorem determines whether a solution exists for a system of linear inequalities. Although general theorem proving is unsolvable, decision procedures for checking the feasibility of a system of linear inequalities exist. Our prototype uses the Boyer-Moore theorem prover[1] for this purpose. [Boyer,1988]

In summary, our approach has the following advantages:

- Generation of stronger loop invariants.
- Symbolic capabilities.
- Flow analysis to handle simpler cases and verification techniques to handle more complicated cases.
- Flow analysis to reduce the number of assertions.
- Cut-points and invariants to reduce the complexity of the theorems.

We start with the intermediate form program, which came from a source program, as input and compute the control flow graph and Def-Use information. The output is a list of the array accesses that cannot be verified.

8.4. Generation of Simple Loop Invariants

Early research on program verification attempted to combine algorithmic and heuristic techniques to generate loop invariants. [Katz,logical] [German,1975] [Wegbreit,1973] [Katz,heuristic] The algorithmic approach generates invariants from assignment statements by using loop exit tests and by propagating predicates. The heuristic approach generates new candidate invariants by strengthening existing invariants, weakening existing candidates, pushing invariants backwards and forwards through loops, using domain specific information, and extracting information from unsuccessful proofs. The invariants generated with the algorithmic approach are usually not strong enough. The heuristic approach starts with assertions provided by the programmer and those generated with the algorithmic approach. Unfortunately, the heuristic approach generates too many invalid candidate invariants.

Template matching is one of the algorithmic approaches. Advantageously, it takes account of the program structure, so it can generate invariants not generated by heuristics. Unlike heuristics, all candidate invariants generated are valid. Template matching can also be fully automated. It is not widely used for general invariant generation, because the many ways to express an algorithm make writing suitable templates difficult. However, we find that the template-matching approach is appropriate when limited to induction variables used in array-subscript expressions. In our approach, an abstracted

---

[1] Linear arithmetic is only one of the capabilities of the Boyer-Moore theorem prover.

program loop is unified against various program templates, and an invariant is generated upon successful unification. Our early experience shows that a small number of templates suffices, and this method suffices for simple invariants, such as generating bounds for the index variables, array bounds, and bounds for pointer values.

8.4.1. Loops in Flow Graph

The input is a program expressed in a Lisp-like intermediate form. For our purposes, we view the program as a program graph. Program-flow graphs often fall into a class called reducible [Aho,Ullman,compiler] flow graphs, which can be partitioned into a hierarchical structure by interval partitioning. Intuitively, an *interval* [Ullman,1986] in the flow graph is a natural loop plus an acyclic structure that dangles from the nodes of the loop. After partitioning the flow graph into intervals, we remove the structure dangling from each loop. If a loop contains nested loops, we abstract each immediate inner loop into one node (i.e., the abstracted loop contains no cycle).

With the abstracted loop, we begin to locate the possible induction variables. We first approximate the candidate set with the scalar variables modified in the loop. The algorithm scans the innermost loops first. Induction variables of the inner loops are eliminated. Variables in the inner loops are also removed from the candidate set. Figure 8.2 contains the details of the algorithm.

Input. A program graph G.
Output. A Candidate Set of Induction Variables.

Interval Partitioning by T1-T2 Analysis. Set $I(v) = \{v\}$. Apply the T1 and T2 transformations. During T1 transformation, in which the arc (v, v) is removed, perform the following steps on the interval $I(v)$.

T1: Delete self-loop (v, v) in G.

T2: If (v, w) is the only edge incident onto w, and $w/ = v$, delete vertex w. For every edge (w, x) in G, add a new edge (v, x) if (v, x) is not already in G. Set $I(v) = I(v) \bigcup I(w)$.

Remove Dangling Structure from Intervals. Construct subgraph L from G consisting of vertices in the interval $I(v)$. Remove those vertices that do not reach the interval header v.

Transform a Nested Loop into a Simple Loop. Determine inner loops embedded in L. For each inner loop M of L, replace the whole loop with a vertex m. Then add arc (v, m) if there is an arc (v, v') for some v in L and v' in M; add arcs (m, w) if there is an arc (w', w) for some w in L and w' in M.

Determine Induction Variables for Each Loop. With loops ordered from inner to outer, for each loop L, determine the set of scalar variables $S(L)$ modified inside the loop. Remove variable x in $S(L)$ if all x's flow-dependent variables are not in $S(L)$. For each inner loop M of L, remove v in $S(L)$ if v is in $S(M)$. Output $S(L)$.

In Fig. 8.3, we demonstrate using the T1-T2 transformation [Aho,Ullman,compiler] described in Fig. 8.2 to determine the intervals. The first T1 transformation produces the interval {2,3,4} headed by vertex 2. The second T1 transformation produces the interval {1,2,3,4,5} headed by vertex 1. A loop is determined by removing the dangling structures (see Fig. 8.2) from an interval. The inner loop is {2,3,4} and the outer loop is {1,2,3,4}. We condense the inner loop into a single node, $loop_2$. The outer loop is then represented as {1, $loop_2$}, as shown in Fig. 8.4. We further condense the outer loop into a single node, $loop_1$.

Then we apply symbolic evaluation to compute the values of the candidate induction variables for one iteration of the abstracted loop. This evaluation is possible because the abstracted loop has an acyclic structure. Symbolic values are assigned for each induction variable at the beginning of the loop. We consider only the symbolic evaluation of linear arithmetic. Other computations generate an undetermined (*bottom*) value. References to variables not changed in the loop generate constant symbolic values. References to variables modified in the loop but not in the candidate set generate *bottom* values. Computations that depend on a *bottom* value have *bottom* value too. The values of the candidate variables of the loop body are collected. Those with non-*bottom* values are considered induction variables. Then these variables are subjected to loop template matching to generate the invariant assertions.

8.4.2. Simple Loop Templates

Our approach makes use the loop structure to generate the loop invariant. A programmer can express computations in a loop in many different ways (i.e., different variable names, and different order of execution). We abstract the computation by symbolic evaluation. We represent common loop structure with loop templates, and unify the result of symbolic evaluation with the templates. Figure 8.5 lists the common recurrence relations that represent loop computations.

	Recurrence Relation	**Invariant Generated**	**Corresponding Loops (examples)**

1	$V_{i+1} = any(V_i, C)$	$V_i = V_0 or V_i = C$	i=N;
			while (B)
			{S1; i=10; S2;}

2	$V_{i+1} = any(V_i, V_i + C)$
	$C \geq 0$
		$V_i \geq V_0$	for (i=0; B; i++);
			i=0;
			while (B1) {
			S2;
			if (B2) i++;
			S3; }

3	$V_{i+1} = any(V_i, V_i + C)$
	$C \leq 0$
		$V_i \leq V_0$	for (i=N; B; i--) S;
			i=N;
			while (B1) {
			S2;
			if (B2) i=i-5;
			S3; }

4	$V_{i+1} = V_i + C_v$ $W_{i+1} = W_i + C_w$ $C_v / = 0 \mathrm{and} C_w / = 0$	$(V_i - V_0)/C_v = (W_i - W_0)/C_w$	for (p=0,q=10; B; p++, q++) S;
5	$V_{i+1} = V_i + C_v$ $W_{i+1} = any(W_i, W_i + C_w)$ $C_v / = 0 \mathrm{and} C_w / = 0$	$(V_i - V_0)/C_v \geq (W_i - W_0)/C_w$	for (p=0,q=10; B; p++) if (B1) q++;

6	$V_{i+1} = any(V_i, V_i * C_1 + C_2)$
	$V_0 \geq 0 \mathrm{and} C_1 \geq 1 \mathrm{and} C_2 \geq 0$
		$V_i \geq V_0$	p=1;
			while (B) {
			S;
			if (B1) p=p*2
			else p=p*2+1; }

7	$V_{i+1} = any(V_i, V_i/C)$
	$C \geq 1$
		$V_i/V_0 \leq 1$	p=N;
			while (B) {
			S;
			if (B1) p=p/2; }

The notation is as follows: V_0 and W_0 are initial values for variables V and W. V_i and W_i are values before an iteration of the loop body. V_{i+1} and W_{i+1} are values after one iteration of the loop body. C, C_1, C_2, C_v, C_w are interval constants--values that do not depend on the iteration of the loop. $X = any(Y, Z)$ means X can assume values of Y or Z in different passes through the loop. Y and Z are patterns to be instantiated. A pattern can have more than one instantiation when matching with the recurrence relation of a loop. For example, the pattern $V_i * C_1 + C_2$ in template 6 (see Fig. 8.5) instantiates with #p*2# and #p*2+1# (see the example program in the rightmost column).

Some approaches use the iteration count in the recurrence solution. However, we find that the iteration count does not describe relations between two expressions directly, as described in templates 4 and 5.

The template-matching method is extensible, as we can easily add more templates to the analyzer. The invariants generated are linear inequalities concerning induction variables; they are sufficiently powerful to describe loop-induction variables and therefore handle most of the loops in common programs. However, a programmer could write programs in a way such that the data dependence is translated into control dependence, or vice versa [Allen,1983], in order to defeat the template-matching mechanism.

8.4.3. Examples

Figure 8.6 identifies one loop, in which scalar variables #i# and #j# are modified and form the candidate set for induction variables. The result of symbolic evaluation on two induction variables #i# and #j# is:

$$i_{l+1} = i_l + 1;\ \ i_0 = 0\ \ j_{l+1} = any(j_l, j_l + 1);\ \ j_0 = 0$$

The unification with template 2 generates two invariants: $i \geq 0$ and $j \geq 0$. The unification with template 5 generates one invariant: $(i-0)/1 \geq (j-0)/1$, which simplifies to $i \geq j$.

```
001   /* Example 1: String Compaction */ 002   char a[100]; 003 004   string_com-
pact() 005   { 006      int i = 0, j = 0; 007      while (i < 100 && a[i] != '\0') 008
if (a[i] != ' ') { 009            a[j] = a[i]; 010            i++; 011            j++; 012
} else 013            i++; 014      if (j < 100) a[j] = '\0'; 015    }
```

The example in Fig. 8.7 has a nested loop. The inner loop modifies variables #thisone# and #first#, which are therefore the candidate set for the inner loop. The outer loop modifies variables #thisone#, #first#, and #top#. After eliminating the variables belonging to the inner candidate set, the candidate set for the outer loop contains only one variable: #top#.

The result of symbolic evaluation is:

$$thisone_{l+1} = thisone_l + 1;\ \ thisone_0 = top + 1$$
$$first_{l+1} = thisone_l;\ \ first_0 = top$$
$$top_{l+1} = top_l + 1;\ \ top_0 = 0$$

The unification with template 2 produces two invariants: $thisone \geq top + 1$ for the inner loop and $top \geq 0$ for the outer loop.

```
001     /*  Example 2:  Selection Sort */ 002     struct datatype { 003         int key, val; 004
} data[100]; 005        006     selection_sort() 007     {  008         int top, bottom, first,
thisone; 009         struct datatype t; 010       011         bottom = 99; 012         for (top = 0;
top < bottom-1; top++) { 013             first = top; 014             for (thisone = top+1; thisone
< bottom; thisone++) {  015                 if (data[thisone].key < data[first].key) 016
first = thisone; 017             } 018             t = data[top];
019             data[top] = data[first]; 020             data[first] = t; 021         } 022     }
```

8.5. Integer-Range Analysis

The integer-range analysis attempts to derive the worst-case bounds that are valid for all possible execution paths for integer variables at each program node. The information propagated along the program graph is restricted to the format $constant1 \leq var \leq constant2$. Information in this format can be computed efficiently, although certain information is lost (i.e., the relation between two variables cannot be expressed). The integer ranges are propagated along with the program; new integer ranges are defined in the assignment nodes; integer ranges are refined by inductive invariants and the array check assertions, and then the integer-ranges are refined by the branch conditions. The propagation and refinement continues until no more changes can be made. Implementation of the flow analysis is discussed in Section 6.5.5. Here we concentrate on describing the usage of loop invariants and array-check predicates to enhance the integer-range analysis.

Notably, the result of the flow analysis is always correct despite the initial ranges. We can start with the most pessimistic range $(-inf, \infty)$ or the most optimistic range $(\infty, -inf)$. In general, the optimistic initialization generates a stronger result (tighter bounds), but the pessimistic initialization has better convergence behavior (see Section 6.1.1 for details). In this prototype, we choose the pessimistic range for the convergence benefit while strengthening our result with the introduction of assertions and loop invariants. The ranges so derived are weak invariants[2] of the program. We can then use them to verify the array accesses.

The array-check predicates become tentative assertions. We use them to refine the range information. If the array check is within a loop, the refinement propagates around the loop and may tighten the ranges. The tightened ranges are sometimes strong enough to verify the array-check predicate. This verification is similar to an inductive invariant proof in which the invariant (the array-access predicate) is first assumed correct and then justified. When the array access is incorrect, it is not possible to justify it. Therefore, this method does not introduce errors (i.e., out-of-bounds array accesses are not accepted) but produces stronger results.

Intuitively, if multiple checks of the same array accesses occur in some path, the dominated checks can be removed. In this analysis, the first array check is assumed correct. The ranges are refined by this assertion and propagated along the program. Some ranges will have new values due to assignment statements, and others are further refined with other assertions. The result is that the array-check assertion will induce some stronger information in other parts of the program, therefore eliminating more redundant array checks than the intuitive method.

8.5.1. Range Propagation and Refinement

Recall the integer-range propagation and refinement problems in Section 6.5.5. The flow values, elements of V, are functions $irange: var \rightarrow (min, max)$, where var is an integer variable and min and max refer to the minimum and maximum integer values that var may have. The *bottom* element of V is defined as $irange(x) = (-inf, \infty)$.

The transfer function f_n of vertex n can be described in two cases. In the first case vertex n is an assignment node. The function computes and updates the range of the assigned variable. For example, given the assignment statement #J:=I*2;# and $irange(I) = (10, 20)$, the new range for #J# is $irange(J) = (20, 40)$. In the second case vertex n is a branch node. The function refines the ranges of variables used in the branch condition.

The confluence function is the worst-case bounds from different incoming edges. For example, if $bound(I) = (0, 10)$ from one edge and (20,30) from the other, the confluenced range is (0,30).

---

[2] The invariants so derived are not strong enough to prove program correctness because we only address index variables.

8.5.2. Special Cases

Two special cases must be handled. In the first case [Harrison,1977] some variables, called *interval* *constants*, are not modified inside the loop. Interval constants are properties that remain unchanged in an interval. With the pessimistic flow analysis, these variables have two paths from which range information comes: one from the loop initialization and the other from the loop iteration. When a variable is not defined in the loop, the range for its values inside the loop body is undefined. When the confluence is computed at the loop header, the worst case (undetermined) is taken. We resolve this situation by ignoring the loop-back arcs when computing values for the interval constants at the loop header.

The second case is quite similar to the first one. As shown in Fig. 8.7, the definition at line 016 of the variable #first# inside the loop is only conditionally executed. There exists a path in the loop body in which #first# is not modified. This path provides no information, hence a *bottom* value is associated with #first#. We solve this by using Def-Use information to refine the ranges at the loop header, i.e., the range for #first# at line 014 is refined from its ranges at line 013 and 016.

8.5.3. Examples

Figure 8.6 illustrates the flow computation. In this program, the loop invariant at line 007 is $i \geq 0 \mathrm{and} j \geq 0 \mathrm{and} i \geq j$. The condition for entering the loop body is $i < 100$, ignoring non-scalar variables. On the first pass, the range for #i# at line 007 from the initialization edge is (0,0), and the range from the loop-back edge is *bottom*. Hence the confluenced range is *bottom*. Using the loop invariant at line 007, the ranges become $irange(i) = (0, \infty)$, $irange(j) = (0, \infty)$. Entering the loop body, $i < 100$ refines the range for #i# to (0,99). The range for #i# shows that the array accesses #a[i]# at line 008 and 009 are correct. Propagating the ranges through the loop body, #i# is updated in both branches, and its range becomes (1,100); also #j# is updated in one of the branches, and its range becomes $(0, \infty)$. On the second pass, the ranges $irange(i) = (0,100) \mathrm{and} irange(j) = (0, \infty)$ at line 007 combine with the invariant $i \geq j$, refining $irange(j)$ to (0,100). Similarly, the ranges propagate through the loop. The analysis detects no changes at the third pass and stops. Unfortunately, the array access #a[j]# at line 009 is not verified. A technique to handle this situation will be discussed in Section 8.6.

The program in Fig. 8.7 can be used to show how the flow analysis eliminates some redundant checks. At line 018, the array access check $0 \leq top < 100$ refines the range for #top# to (0,99). The range propagates to line 019 and validates the check for #data[top]# there. Similarly, the array access check for #data[first]# at line 019 validates the one on line 020. A more complicated case occurs at line 019 where the value of #first# is the value of #thisone# at line 015 or that of #top# at line 013. The range for #top# is (0,97).

The access check for #data[thisone]# at line 015 constrains the range for #thisone# to (0,99). Similarly, the access check for #data[first]# at line 015 constrains the range for #first# to (0,99). The confluence of these three ranges is (0,99), thus validating the access #data[top]# at line 019.

Figure 8.8 shows another case that flow analysis fails to handle completely. The proof of the access #str[i+j]# at line 012 requires symbolic evaluation. The range for #len#, after line 008, is (0,10). At line 012, after refinement by the branch conditions $i < 100 - len$ and $j < len$, #i# has the range (0,99), and #j# has the range (0,9). Taking the worst-case irange, the range for the expression #i+j# is (0,108). Obviously, it is not strong enough to prove the required irange (0,99) for the array #str#. This limitation will be handled by verification techniques.

```
001 /* Example 3: Find pattern in string */
002 string_index()
003 {
004 char str[100], pattern[10];
005 int i,j,len;
006
007 for (len=0; len < 10; len++)
008 if (pattern[len] == '\0') break;
009 /* len is the length of the pattern string */
010 for (i = 0; i < 100 - len; i++) {
011 for (j = 0; j < len; j++)
012 if (str[i+j] != pattern[j])
013 break;
014 if (j == len) break;
015 }
016 if (j == len)
017 return i;
018 else
019 return -1;
020 }
```

8.6. Verification-Condition Generation

We use verification techniques to complement the limitations of flow analysis. In flow analysis, we use two integers to denote the upper and lower bounds of the range. This notation is not always strong enough, e.g., the irange generated from $X/=1$ is $(-inf, \infty)$. Similarly, when two non-overlapping ranges merge together, the gap in between is lost. As seen in the previous section, the flow analysis is not able to handle symbolic constants (Fig. 8.8), and it loses some information during the sequential evaluation of the constraints (Fig. 8.6). The VCG should handle these cases.

The VCG (see Fig. 8.9) collects assertions that are valid at the point of array access check and passes the collected assertions to the theorem prover. Sequential evaluation is not a problem here because the Boyer-Moore theorem prover has symbolic capability. Because general verification-condition generation along with invariant generation can produce very complicated assertions, we still limit the assertions generated to linear inequalities.

We include the acyclic reaching definitions (see Section 6.5.4) at the loop header for each variable modified in the loop. This definition kills definitions from initialization or loop iterations, thus serving as the unique definition in the loop. The use of acyclic reaching definitions will not weaken the assertions generated because the cyclic dependence has already been handled by the inductive invariants.

To avoid generating a verification condition that applies to the whole program, we apply data-flow slicing (see Section 7.3) to include the related definitions and assertions. The cost of verifying each assertion remains approximately constant regardless of the program size.

Input. The array access check C and the corresponding vertex n.
Output. The verification condition $A\mathrm{and}D \Longrightarrow C$ where A is the collected assertions and D is the collected definitions.

Let $use(expr)$ be the function returning a list of variables reference in $expr$. $areach(var, node)$ returns the acyclic reaching definitions of var at $node$.

1. Rename Induction Variables.
For each loop L and each variable V modified in L, assign a new definition *V-L* at the loop header, so that no definition can depend on itself. New acyclic reaching definitions $areach$ are computed using flow analysis.

2. Compute the dominators for the program graph.
Use flow analysis.

3. Collect Assertions.
I is the loop invariant or the branch condition associated with nodes that dominate n. If $Use(I)$ overlaps with $Use(D) \bigcup Use(C) \bigcup Use(A)$, include I into A.

4. Collect Definitions.
For all $uinUse(D) \bigcup Use(C) \bigcup Use(A)$ where $areach(u, n) = \{d_1, d_2, \cdots\}$, include the assertion $u = d_1 oru = d_2 or \cdots$ into D. Repeat this step until no more additions to D.

5. Repeat step 3 and 4 until convergence.

8.6.1. Examples

In Fig. 8.6, the array access check for #a[j]# at line 009 is $0 \le j < 100$. The flow analysis has verified $0 \le j$, but is unable to handle $j < 100$. We use the VCG to generate a verification condition for this case:

$$(j{:}9 = j{:}7 \text{and} i{:}7 < 100 \text{and} 0 \le i{:}7 \text{and} 0 \le j{:}7 \text{and} j{:}7 \le i{:}7) \Longrightarrow j{:}9 < 100$$

The notation $j{:}9$ refers to the value of j at line 9. The VCG applies data-flow slicing. It starts with the check $j{:}9 < 100$ in which $j{:}9$ is used. There is only one definition for j ($j{:}7$) reaching line 009; it produces the condition $j{:}9 = j{:}7$. The VCG then looks up the loop invariants and branch conditions that involve $j{:}7$ or $j{:}9$. It identifies and includes into the condition the loop invariants $j{:}7 \le i{:}7$ and $0 \le j{:}7$ at line 007. The set of variable definitions includes $i{:}7$ now. The VCG identifies and includes one more invariant, $0 \le i{:}7$ and one branch condition $i{:}7 < 100$ involving $i{:}7$. The addition of these predicates does not add new variable definitions to the system, therefore the data-flow slicing stops. The predicate is then transformed into proper format and proved using the Boyer-Moore theorem prover. [Boyer,1988]

In Fig. 8.8, the array access #str[i+j]# at line 012 requires symbolic evaluation. The VCG collects related predicates and definitions into the verification condition. Its essence is:

$$0 \le len \le 10 \text{and} i < 100 - len \text{and} j < len \Longrightarrow (i + j) < 100$$

8.7. Implementation and Results

Figure 8.10 shows the execution time on a lightly loaded SPARCstation-2 for processing the example programs described in Figs. 8.6 through 8.8 plus a binary-search and a tree-sort program. The program Combined, which combines all five test programs, serves to indicate the cost of the algorithm applied to larger programs. The execution times (CPU time) are averages measured in seconds for those runs that do not invoke garbage collection.

	Invariant Generation	Flow Analysis	VC Generation	Number of Lines	Number of Integers	Number of Checks
Compact String	0.05	0.18	0.01	19	2	3
Selection Sort	0.05	0.36	0.00	25	4	6
String Match	0.05	0.45	0.01	24	3	3
Binary Search	0.03	0.46	0.02	24	4	3
Tree Sort	0.11	1.4	0.00	53	6	18
Combined	0.25	5.4	0.35	145	19	33

The programs tested are small, and the sample size is small too. We cannot draw comprehensive conclusions from them. However, the followings are our conjecture about the performance of the array bounds checker based on the preliminary result.

- The invariant generation time is essentially linear to the number of induction variables (program size).
- The execution time of a pass in the flow analysis grows linearly with the number of variables times and the program size. The number of passes is the depth of the program graph which increases slightly with the program size.
- The total execution time of the VCG depends on how many array checks are not verified by flow analysis. The time needed to generate a VC depends on the number of assertions, invariants, predicates and definitions collected. We believe most VCs are simple and therefore the cost of VCG is small.

8.8. Discussion

With the use of data-flow information, we devise a practical way to generate simple loop invariants to check array accesses statically. The complexity of loop-invariant generation is roughly proportional to the number of loops in the program. With loop invariants, we strengthen flow analysis in two ways: we require fewer passes are required and validate more checks. We employ program-verification techniques for those checks not validated by range analysis. In generating the assertions, data-flow information is again used to include only the relevant constraints and definitions; therefore, the time to verify an array access is dependent only upon the complexity of its recurrence relation and is independent of the program size. Therefore, our approach can deal with large programs.

Our approach is extensible due to the use of templates. We have produced a small library of templates corresponding to common loop structures. New templates for new applications can be added. Suzuki indicates that his approach [Suzuki,1977] has problems dealing with traversal of data structures ending with a special value, e.g., null-terminated strings. We propose creating new templates to deal with such comparisons in a program.

The techniques used in this dissertation are useful for other applications. Flow analysis produces a lower bound and an upper bound for integer variables and expressions. When the bounds are equal, we know the variable has a constant value. When the range of the expression in a switch-statement is small, a compiler can generate a more efficient indirect jump-table for it. Furthermore, branch conditions involving integer variables can sometimes be simplified. Branches not taken can then be removed. The techniques to recognize induction variables, which make use of local information (natural loops, data dependence, candidate set of induction variables), can make the pure abstract interpretation approach [Ammarguellat,1990] more efficient by eliminating unnecessary computations. The recognition of recurrence relations, which may be expressed in different statements or in different orders of execution, finds many applications in parallelization of numerical computations. By assigning a unique identity to each definition and each loop induction variable, we obtain an acyclic description of the Def-Use chain, which allows a straightforward verification-condition generation, and reduces the complexity of the assertions generated this way.

Our method is not without shortcomings. The flow analysis ignores certain control-flow information (when information from incoming arcs merges, we do not distinguish their flow paths) and uses propositional logic and linear inequalities to express assertions. This limits our analysis to proofs expressible in this paradigm. We do not consider many algebraic theorems such as $sqrt(4) = 2$, $sin(0) = 0$, and conversions between floating point and integers. Values read from input or files when the programs are executed (usually available in form of input assertions) are not used for the proof. Furthermore, the integer-overflow problem is not considered here (i.e., 32767 + 1 overflows in a 16-bit machine). It seems that only the most general program-verification techniques can solve all such problems and, furthermore, we have found that these problems do not occur in most of the programs we investigated with respect to array access checks.

CHAPTER 9

Detecting Malicious Code and System Vulnerabilities

9. _9

In the previous chapter, we presented the implementation of the filter tool MCF. In this chapter, we demonstrate how to use the tool to detect malicious code and security-related vulnerabilities. We first present some malicious code and then provide a taxonomy for them. The purpose of the taxonomy is to collect and analyze typical characteristics of known malicious code, and predict the characteristics of future malicious code. Based on these characteristics, we determine the tell-tale signs that generate small program slices and are therefore suitable for discriminating malicious from benign programs.

Section 9.1 explores the motivation of a malicious-code writer. Section 9.2 examines how this motivation can be realized through malicious programs and then defines a skeleton for such programs. Section 9.3 examines some known system vulnerabilities. Section 9.4 expresses the malicious-program skeleton in terms of tell-tale signs and discusses the use of MCF for their detection. Section 9.5 examines a system program that involves an extensive amount of authentication code. Section 9.6 examines a game program that contains nothing security-related. We can see that a security analyst can save a lot of work examining non-security related programs by using MCF. Section 9.7 describes the problems in examining these programs.

9.1. Motivation for Writing Malicious Programs

A malicious-program writer's objectives usually fall into the following categories: financial gain, personal benefits, revenge, and curiosity and fun. In all cases, malicious code affects the operation of the attacked system, causing loss of money and computer services, and loss and disclosure of information. We list the categories and some of the objectives in each category below:

Financial Gain

- Increase the salary field in the employment record.

- Change the social security status to obtain benefits.
- Forge or delete transactions in a banking system.
- Move a cent from each account in a bank to a specified account.

Personal Benefits

- Change the grades in the student record.
- Read tomorrow’s exam questions.
- Read other students’ assignments.
- Steal the latest version of proprietary software.
- Steal corporate secrets.

Revenge

- Bring down the computer system.
- Increase the load, and therefore the response time, of a system. If the system is a stock-exchange system, the transactions will not be carried out fast enough. If the system is a life-support system in a hospital, patients may die.
- Eliminate someone’s record (e.g., employment record, student record) to cause trouble.
- Overwrite the entire system at a certain time.
- Harass other users by changing their .login files.

Curiosity and Fun

- Look into the IRS system to see how much tax George Bush pays.
- Browse the system for interesting files.
- Challenge existing computer-security mechanisms (i.e., obtain unauthorized access to a system, become superuser, read/modify protected files, etc.).
- Experiment writing malicious programs for fun, e.g., viruses, worms.

9.2. Typical Malicious Programs

Although the above objectives are incomplete, they all exhibit some anomalous behavior: reading protected files, modifying protected files, and obtaining unauthorized privilege. We further refine their anomalous activities into six steps of performing malicious actions. This analysis identifies the commonality of malicious code and develops tell-tale signs to detect them. We hope that this model will also fit future malicious code.

(1) **Gain access to the system.** A malicious code must be installed in a system before it can be activated. It may be installed by an insider who has the appropriate privilege. As a Trojan horse, it may be installed by causal users who obtain the malicious code from a public bulletin board. As a virus, it may attach itself to a user's diskette when the user accesses an infected machine. To a lesser extent, an outsider who does not have direct access to the system can install malicious programs through known OS bugs or flaws [Spafford,vulnerabilities,1992] in protection settings (protection states).

(2) **Obtain higher privilege / Retain current privilege.** Once a program is installed in the system, it may belong to a particular user in the system, but it may not have sufficient privilege to perform the malicious action. The malicious program may want to retain the privilege beyond the termination of the current process, so that the malicious action can be performed at a later time.

There are many ways to expand the privilege in a UNIX system. As mentioned in Step 1, the malicious code can exploit bugs in OS and privileged applications, or incorrect protection settings. The protection settings of a UNIX system can be legitimately altered directly or indirectly. With the direct methods, the file access mode, setuid bit, setgid bit, the file owner id, and the group id can be changed by the system calls chmod, chown, and chgrp, respectively. The indirect method is to change the files or databases containing authorization information (e.g., /etc/passwd, /etc/exports, /etc/hosts.equiv, and ˜user/.rhosts).

The privilege can be expanded by exploiting the indirect flow of privilege in UNIX. For example, you can gain root access if you can modify a file that will be run by root. By obtaining read access to /dev/kmem, /dev/mem, you can read the raw password from the memory space of the login process. Similarly, read accesses to the /dev/tty* devices can collect passwords from logins. If writes to /dev/mem or /dev/kmem are granted, you can zero the userid field in the kernel process table and upgrade a process to root privilege. In other cases, if you can modify /etc/aliases (which sendmail interprets), you obtain the privilege of sendmail.

The direct holes may be closed by carefully examining the protection mode of security-related system files. The indirect holes are harder to close because a thorough understanding of the interaction of various components in the system is required. The COPS package [Cops,Farmer] detects direct holes, and the Kuang [Kuang] package identifies some of the indirect holes.

(3) **Wait for the proper condition or look for certain patterns.** A malicious code starts when certain conditions are met. For example, a PC EXE virus only infects EXE files in the system. Some viruses will not propagate most of the time, so that their propagation is slower and therefore less noticeable. A time bomb activates at a certain time (e.g., Friday the 13th). A logic bomb activates when certain combinations are detected (e.g., when the system load average is 12.34). Malicious programs that steal information search for particular keywords or strings in files.

(4) **Perform the action.** The actions depend on the objectives of the malicious-code writer. Although many different actions are possible, their implementations typically include file accesses, file modifications, and executions of other commands.

Virus replication can be viewed as the modification of executable programs. The worm replication is the remote execution of a worm segment. Malicious programs that steal information just read the relevant files and send them back to the writer, e.g., by electronic mail, by a network connection, or even by covert channels. Malicious programs aiming to get privilege usually modify system files; programs introducing trapdoors modify executable programs that have root privilege. Malicious programs requiring time-delayed damages need to create another process to commit the damages. For denial-of-services attacks, the malicious code may monopolize the CPU, consume a lot of memory, or even crash the system.

(5) **Clean up.** To avoid detection, a malicious programmer may remove the origins of the malicious code from the system. If the goal was to obtain some information, the programmer will not want to be traced from the returning information.

Before activation, the malicious program may avoid obvious appearance. After activation, it eradicates itself after the damage. Viruses may restore the original executable program. For example, the Internet worm avoided leaving information in the file system by unlinking itself. More sophisticated malicious programs may want to reverse the audit information from the system. If the audit privilege has been obtained in Step 2, it is more desirable to suspend the audit trail while the damage is being performed.

(6) **Repeat steps 1 to 5.** Malicious programs, such as viruses and worms, may terminate when something has been done or they may decide to wait for another chance. Once they propagate to other systems, they will start from Step 1 again.

Although conventional viruses and worms replicate blindly, target-seeking viruses and worms, which replicate in a controlled way, can be built. A malicious program seeking specific information might migrate from one system to another to search for the desired information; only one copy of the malicious program is maintained to make detection harder. Similar to a multistage rocket, the malicious codes may carry themselves to different, typically more protected, environments. Through this method, the malicious code attacks highly protected systems or systems the intruder cannot access directly.

To attack a system shielded from the outside by a network gateway, a malicious program needs to infect the gateway first and then jump from the gateway to the desired system. To infect an embedded system, in which the programs are usually stored in ROM, a malicious program needs to infect the development system first.

Future malicious code will be more intelligent than it is today. It might have artificial intelligence to determine which information is worthiest or to which system it should migrate. This kind of malicious program will be smart enough to avoid detection by dynamic analyzers and intrusion-detection systems. However, the complexity of such malicious code is high enough that certainly some tell-tale signs will be apparent. The sheer size of these malicious codes will only make static detection easier.

We studied viruses in Section 2.5.1 and time bombs in Section 2.5.2. We analyzed a real UNIX virus and discussed its detection in detail in Section 2.6. Now we analyze some example malicious programs.[1] We use these examples to illustrate how the six steps mentioned above are applied towards analysis of the malicious code. Section 9.4 discusses their detection.

---

[1] We have no access to real malicious-code, therefore we develop our own examples based on known malicious programs. These examples are realistic.

9.2.1. Trojan Horse

The Trojan login program is usually advertised as some enhancement to the existing login program (e.g., using shadow password).

(1) It is copied to the system by the administrator.

(2) It is installed to the /bin directory as a root-setuid program.

(3) An outsider enters the system using a bogus userid, for which a password is not required by the Trojan login program.

(4) A root-privileged shell is created for that particular login.

(5) The login program does not write the bogus login to the log file, so the bogus login will not show up with system-administration programs.[2]

9.2.2. Salami Attack

The malicious program is a paycheck printer that prints a check of extra amount.

(1) The program is written and installed by a programmer (insider).

(2) The program is allowed to access the checks and employee databases at the end of the pay period, i.e., started in the biweekly batch job.

(3) The triggering condition is when the employee name is Mr. X.

(4) The action is to deduct a small amount (e.g., one dollar) from each employee's account, and put the sum to the amount printed on Mr. X's check.

(5) The malicious code evades detection by balancing the payroll expenses.

9.2.3. Sniffer

The sniffer is a modified telnet program that monitors the session traffic and collects "interesting" information. Although this program already has the user's privilege, it still

---

[2] It could show up in a command-log file.

cannot obtain the encrypted information.

(1) The program is provided by a vendor, and someone in the development team of the vendor wants to collect information.

(2) The program is activated when a user starts a new telnet session.

(3) The program monitors the telnet traffic for certain strings. If the intruder wants to collect marketing information, the trigger string could be “market”. If the intruder wants to collect passwords, the trigger string could be “login”, “passwd”, or “password”.

(4) When the trigger string is found, the program collects the next 10 bytes for collecting a password or the next 1000 bytes for collecting the marketing information. After the information is collected, the program uploads the information to an anonymous ftp server.

(5) The intruder avoids being identified by sending the information to a public file server, which anyone can access.

9.2.4. Multistage Launcher

This mechanism carries a malicious program into a specified location (system). It is similar to the replication of viruses, but it replicates in a controlled way and has a target. It has no specific malicious action except propagating to more secure systems.

(1) The multistage malicious program is installed as a Trojan ‘ls’ program in the /tmp directory by an insider.

(2) Users working in the /tmp directory may execute the Trojan ‘ls’ program accidentally.

(3) After invocation, the malicious program determines whether it should migrate to a remote system accessible by the current victim (i.e., whether the remote system is closer to the target machine).

(4) If the malicious program migrates, it copies itself to the file /tmp/ls on the remote machine.

(5) The program avoids detection by maintaining one copy of itself all the time.

(6) The program repeats Steps 1 through 5 until the specified machine is reached. The triggering and the action of the malicious code is programmable. For example, it can be programmed to deliver the following malicious code into a development system.

9.2.5. Development System Attack

This attack is aimed at embedded system. The malicious program in this attack has two stages. The first stage gets into the development system and installs the second stage of this in the weapon system. The second stage malicious code creates a blind spot in the firing-control component in an embedded weapon system.

(1) It uses the multistage launcher to get into a development system.

(2) It is executed by a system administrator.

(3) The action is triggered when the program has the privilege to modify the library file (e.g., the C library /usr/lib/libc.a in UNIX).

(4) It changes the sin() function in the library, so that $sin(x) = sin(45)$ when $44 < x < 45$. The effect is that the firing system (the gun activator) can never aim at an angle between 44 and 45, thus it provides the enemy with a safe direction of attack.

(5) The program eliminates itself once the library is modified.

9.2.6. Ferret Program

The ferret program is a malicious program that scans the file systems for certain information. This information could be the latest software or some confidential documents. Steps 1 and 2 reports nothing of significance.

(3) The action is triggered each time the program is executed.

(4) The program scans the current file system for some information by looking for keywords. Once the information is found, a network connection is established to send back the collected information.

(5) To avoid noticeably slowing down the computer, it randomly picks up one file to scan each time.

9.2.7. Overloader

The overloader is a malicious program that imposes glitches in the system by using excessive CPU times or grabbing excessive memory. Its goal is to slow down the system to deny services. Since nothing is interesting about Steps 1 to 4, we discuss Step 5 only.

(5) The action is to create (fork) a large number of processes, and each process requests the maximum amount of memory and randomly accesses the memory. These processes will exhaust all main memory and slow down the system a lot.

Finally, various malicious programs can be combined to form new ones. So far we have assumed all the steps are performed by the malicious program. In practice, some of the steps may involve a human being. For example, an intruder may use a malicious program to obtain root privilege. After obtaining this privilege, the intruder may login to the system directly to perform the cleanup.

9.3. Typical UNIX Vulnerabilities

We mentioned the access-control logical error in the UNIX chown system call in Section 2.5.3 and the initialization bug in the ftp authentication routine in Section 2.5.4. Here, we present other types of known UNIX vulnerabilities: the pointer-overflow program in fingerd, the race condition in rdist, the unaware-file-accesses potentials in comsat, the existence of a hidden path to execute a program from sendmail, and the ability to replace the library function called by root-privileged programs using an obscure feature in ld.so. The next section discusses how to detect these problems using our techniques.

9.3.1. Finger Daemon (fingerd)

The Internet worm hits the finger daemon (fingerd) by overflowing a buffer that was allocated on the stack. The overflow is possible because fingerd used the library function gets(), which does not check for overflows. The allocation of the buffer on the stack allows the stack frame to be overwritten. The overwritten stack frame contains a return address to a small procedure entered in the buffer area through gets(). This small procedure provided by the Internet worms forks an interactive shell. The fingerd connects to the network in duplex mode and runs as root. The intruder can use the interactive shell to do almost everything.

The fingerd has another bug that allows an intruder to read protected files without proper privilege. The fingerd, running as root, prints the content of the .plan file of the person being fingered. Therefore, an intruder can symbolic-link his .plan file with a protected file and then run finger, which invokes fingerd, to print out the content of the protected file. This bug was fixed by first checking that .plan is not a symbolic link before opening the file. However, this fix can be circumvented if the intruder links the .plan file during the period after the check has finished and before the open() system call executes.

9.3.2. Remote File Distribution (rdist)

The rdist program checks the access to a particular directory. If access is permitted, rdist writes the requested file into that directory. However, the checking and the actual file creation are not atomic. By manipulating the timing, an intruder can provide a path with proper permissions for the access check and another the path (using a symbolic link to point to a system directory) for the file transfer.

9.3.3. Mail Notifier (comsat)

The utmp file records information about who is currently using the system. Whenever a user logs in, login fills in the entry in /etc/utmp for the terminal on which the user logged in. /etc/utmp is owned by root but is world writable. Anyone who has an account on the system may modify /etc/utmp. If the system enables tftp, /etc/utmp can be modified from other systems.

The mail notifier (comsat) is the server process that waits for reports of incoming mail and notifies users who have requested to be told when mail arrives. Comsat listens on a datagram port associated with the biff service specification (see services in Section 5 of Unix man pages) for one-line messages of the form user@mailbox-offset. If the user specified is logged onto the system and biff services have been turned on, the first part (10 lines) of the mail is printed on the user's terminal. Comsat reads the file /etc/utmp to determine the appropriate terminal to which to write the mail message. Furthermore, comsat is run as root.

An intruder can modify the terminal field in an /etc/utmp entry to /tmp/x and link it to a system file, e.g., /etc/passwd. Then he can turn on the mail-notification service and send himself mail. Comsat will write the first few lines of the mail message to the target file. If the target file is the password file, the hacker can supply a bogus password entry in the mail he sent himself.

9.3.4. Dynamic Link Editor (ld.so)

SunOS system programs are dynamically linked with the shared library at load time. A user may specify an alternate search path for the dynamic libraries by setting the LD_LIBRARY_PATH environment variable. A user may provide an alternate library by making one in his own directory.

The /bin/sync program is dynamically linked. /bin/sync is the default shell for the account sync, which has no password but runs with root privilege. A user can execute /bin/login from his login shell. /bin/login has a -p option, which preserves any existing environment variables, including LD_LIBRARY_PATH.

An intruder can break into the system by setting the LD_LIBRARY_PATH environment variable to his directory and using /bin/login -p sync to invoke /bin/sync. /bin/sync, which is linked with the library containing malicious code located in the intruder's home directory, is executed with daemon privilege. The malicious code is activated when /bin/sync makes any system or library call.

9.3.5. Configurable Mail Router (sendmail)

Sendmail is a configurable, rule-based, mail-delivery program. The rules are stored in the configuration file /usr/lib/sendmail.cf, which controls the behavior of sendmail. However, the sendmail program can be told to use an alternate configuration file with the -C option (i.e., sendmail -Calternate_file). Sendmail is also a root-owned setuid program.

The following is a line from /usr/lib/sendmail.cf, which uses the program /bin/mail to deliver local messages:

```
Mlocal, P=/bin/mail, F=rlsDFMnm, S=10, R=20 A=mail -d u
```

It is very easy to copy the file to mysendmail.cf and change the line to

```
Mlocal, P=/tmp/getprivilege, F=rlsDFMnm, S=10, R=20 A=mail -d
```

By starting sendmail -Cmysendmail.cf to deliver a local message, sendmail becomes root and executes the malicious agent /tmp/getprivilege with root privilege.

9.4. Using the Filter Tool

The detection of common malicious activities involves identifying the activities involved in Steps 1 to 6 described in Section 9.2. The detection of UNIX vulnerabilities (security properties) includes identifying certain common mistakes in writing privileged applications. We can also extend the tool to verify certain authentication procedures. Banks and commercial firms are usually more interested in data integrity than in secrecy problems. Therefore, we also want to evaluate some integrity problems. Although our discussion is limited for simplicity to the UNIX operating system, it applies to other operating systems as well.

We first present some generic tell-tale properties and then demonstrate how the security analyst can use these generic properties to locate "interesting" sections of a program for further examination. In the following sections, we also describe how the analyst can use the tool to dissect and examine the program.

9.4.1. Generic Tell-Tale Properties

The security analyst may apply the following list of tell-tale properties to identify unusual behaviors of the program. Usually these properties lead to small program slices. Interestingly, the slices for user programs are empty, and the slices for a system program are identical. The work required by the analyst is not proportional to the number of tell-tale properties.

- **File Read.** This includes the slicing for the open() system calls. The list of files being read will show what kind of information the program may access. Strange accesses to /dev/* should be detected.
- **File Write.** In addition to the open() system call, it includes the uses of create(), link(), and unlink() system calls because a file modification can be simulated by deleting and creating a file. The files written to should be checked against a list of important system files (e.g., files the /vmunix, /etc/passwd, /etc/aliases, /bin/*, /usr/bin/* files).[3]
- **Process Creation.** A malicious program uses the fork() system call to create processes. A denial-of-service malicious program may put a fork() system call in a loop to create a large number of processes.[4]
- **Process Execution.** A malicious program may create another process to perform the malicious action, so we check which other programs are invoked and examine them. Typical sequences are a fork() system call followed by an exec() system call, and the system() and popen() library calls.
- **Network Accesses.** Malicious programs can use the network to send information back to the writer. We will slice for the network system calls, e.g., socket, connect, send.

---

[3] Symbolic links to these files could exist. We depend on intrusion-detection systems to notify the system administrator when these links are made.

[4] The number of processes created is limited by the maximum number of processes per user in some UNIX systems.

- **Change of Protection State.** We slice for the change of protection-states system calls, e.g., chmod and chown. It is rather anomalous for normal programs to use these system calls.
- **Change of Privilege.** We slice for the setuid() and setgid() system calls.
- **Time-Dependent Computation.** We find out how the time is used in the program. A forward slicing on the gettimeofday() system call shows all variables that contain time-dependent values. We will slice again for the statements depending on some time-dependent values.
- **Input-Dependent System Call.** It is a refinement of the file open tell-tale sign. Some UNIX applications have data-flow paths from a read() system call to a open() system call. That means a user can probably control which files these applications write by supplying certain inputs.
- **Anomalous Data Flow.** This relates to possible bugs in a program. Some detectable anomalies include consequent definition of variables without any usage in a path, use of undefined variables, and branch testing that depends on a constant value.
- **Anomalous Interprocedural Data Dependence.** We compute the summary data-flow information for each procedure and create a data-dependence graph. The nodes are procedure, and the edges from node A to node B signify that procedure A uses some variables modified by B. Malicious code that does not mix well with the original program will show up as a disconnected component in the summarized data-dependence graph.
- **Race Conditions.** Race-condition bugs occurred in the rdist and fingerd program. The file accesses are validated before the files are read or written. In both cases, an intruder can relink the file being read or change the directory being written into to read or write other files. This race condition is characterized by an access() system call preceding by an open() system call.
- **Well-Behavedness.** This involves two checks: the check that dereferenced pointers contain valid addresses and that pointers do not overflow. We also look for uses of the gets() library call which do not limit the size of the input string.
- **Limited Changes.** This detects whether information is changed. For example, the telnet program should not pass information back and forth without modification. The chfn (change finger name) program should only modify the database information field of the password entry. We slice the program between the corresponding read() and write() system call for the modification of the values. This may also apply to analyzing the paycheck program.
- **Authentication.** We want to find out how authentication is performed. We slice for the conditions that are true for the authentication to be granted.
- **Internal State of Authentication.** Many computations should be stateless (e.g., the paycheck printer and the authentication of ftp program). A stateless procedure is one that only depends on its input parameters and not on any global or static variables. This information can be derived easily using the data-flow information.

9.4.2. Detecting Malicious Programs

Recall the malicious programs mentioned in Section 9.2; they are detectably malicious using the generic tell-tale signs. The Trojan login is detectable by examining the authentication code, i.e., password checking. There is a path starting from the entry point to the privilege-granting part without password checking. The analyst will need to locate the privilege-granting setuid() system call and then slice for the authentication code. With the Trojan horse, the analyst should identify a path to the setuid() system call without passing through the password-comparison code.

In the salami-attack example, an important property is the integrity of the amount printed on the check. We can slice for the amount, which should come from the employee database and implemented by a read() system call. Any modification in the path is very suspicious. Furthermore, the paycheck printer is a filter program in which individual transactions are independent of each other. Therefore, the printer loop should have minimal state information. Some state information, such as counting how many checks have been printed, is expected. By intersecting the slice for internal states and the slice for the amount printed, the malicious code becomes obvious.

The sniffer program keeps information without authorization. The security analyst needs to slice on the communication streams, in which data read should be written out without modification or copied somewhere else. This program executes the ftp program to deliver the information, which is easily detected by looking for the exec() system call.

The multistage program executes rcp or rsh to transfer itself from one machine to another. The execution of rcp or rsh is discovered by the “process execution” tell-tale sign.

The development-system attack program is carried by the multistage launcher. Since this program damages a system by modifying its functionality slightly, there is no effective way to identify it (because so many ways to change a functionality and there are so many functionalities in a system). However, it is still detectable because of the launching section and the library modification.

The ferret program scans the file system for certain information. This program opens many files--an activity detectable by slicing for file accesses. It sends information back through Internet connections. We slice for the network accesses and see how information is shipped back.

In the overloader example, we can slice for the fork() system calls. It would be suspicious if the fork() system call is embedded in a loop that is not interrupted by any exit() or exec() system call.

9.4.3. Examining System Programs

The bugs mentioned in Section 9.3 are also detectable using the generic tell-tale signs. The fingerd overflow problem is identified by the well-behavedness checker. The rdist race condition is detectable by looking for the sequence access() followed by open() using control-flow slicing. The comsat problem is revealed by the file access slice, which indicates that the file written to comes from the #/etc/utmp# directly. Further slicing on the access of #/etc/utmp# shows that its content is not validated. By slicing on the exec() system call, we find that sendmail executes other programs. By slicing on the ld.so program, we find that it opens a file specified by the LD_LIBRARY_PATH environment variable. This information makes the use of the dynamic library apparent to the security analyst.

9.5. Example: Analysis of login.c

We showed the detection of viruses and time bombs in Chapter 2. Here we demonstrate how to handle some benign programs (e.g., analyzing the login program). The following section shows the analysis of the UNIX BSD 4.3 /bin/login program. The source is attached in the Appendix.

9.5.1. Determining Slicing Points

- **File Read and Write.** We first locate the open() system calls, and then use approximate backward data-flow slicing to determine the value of the filename arguments. login has five open() system calls. #/etc/nologin# and #/etc/motd# are read. #/etc/utmp#, #/usr/adm/wtmp#, and #/usr/adm/lastlog# are modified. The slices for each of them follows:
  - 53 char nolog[] = "/etc/nologin";
    231 nlfd = fopen(nolog, "r");
  - 306 f = open("/etc/utmp", O_WRONLY);
  - 312 f = open("/usr/adm/wtmp", O_WRONLY|O_APPEND);
  - 56 char lastlog[] = "/usr/adm/lastlog";
    317 f = open(lastlog, O_RDWR);
  - 460 mf = fopen("/etc/motd", "r");
- **Process Execution.** We find one execlp() system call; the program executed is stored in #pwd->pw_shell#. We need to slice for the value of #pwd#.
  - 397 execlp(pwd->pw_shell, minusnam, 0);

- **Explicit Network Accesses.** Login has no network accesses. Indirect access through remote login is possible, but it is not important because login has no control over it.
- **Change of Protection State.** Login uses chown() and chmod(). We identify the following statements. They use #ttyn# and #pwd#. The slice for #pwd# is grouped with the that of process creation.
    - 339 chown(ttyn, pwd->pw_uid, TTYGID(pwd->pw_gid));
    - 342 chmod(ttyn, 0620);
- **Change of Privilege.** Login uses setuid() and setgid(). Again, it depends on #pwd#. The value of #pwd# is identical to that of the process creation and the protection-state tell-tale signs, so we can do these slices at the same time.
    - 343 setgid(pwd->pw_gid);
    - 348 setuid(pwd->pw_uid);
- **Time-Dependent Computation.** We use control-flow slicing from the time() system call to the end of the program. We then use pure forward data-flow slicing to find variables containing time-dependent values. Next, we use pure forward control-flow slicing to the set of statements executed conditionally with a time-dependent value. We identify one time() library call, but no statements executed depending on the value of time. The time records the login time of a user.
    - 304 time(&utmp.ut_time);
- **Input-Dependent Computation.** We identify whether any input values affect some security-related system calls. We try to locate paths leading from a read() system call to a open() system call. No such paths are found.
- **Independent Modules.** The program has a very flat call structure. #main# calls #doremotelogin#, #getloginname#, #rotterm# #showmotd#, #stypeof#, #doremoteterm#, and #setenv#. #doremotelogin# calls #getstr#.

  The program has three disconnected components by considering aggregated data flow at the procedural level, as shown in the following:

    - main, doremotelogin, getloginname, rotterm, showmotd, stypeof, doremoteterm, getstr
    - timedout
    - catch

      The first one is the main body of the login program. The other two are the signal handlers implementing timeouts. After examining #timedout# and #catch#, no malicious code is found.
- **Race Condition.** We use control-flow slicing to narrow the search in the program between an access() and an open() system call. Then we use backward data-flow slicing for the arguments in the open() system call. Only one access() is found, and its argument #qlog# is not used by any open() system call. Therefore, login does not have this race condition.
    - 54 char qlog[] = ".hushlogin";
      316 quietlog = access(qlog, F_OK) == 0;

- **Authentication.** We combines the slicing for authentication with that for #ttyn# and #pwd#. We want to find out under what conditions the protection-state changes are granted. The authentication slice is significantly larger than the slice for #ttyn# and #pwd#. For this particular example, the size reduction is very small, mainly because the whole login is written for authentication.
- **Examining Internal State.** Statements 183 to 288 are the authentication loop -- if the authentication fails, the program obtains another userid and password and retries. We try to determine the state variables of this loop. A variable is a state variable if it is also an induction variable (i.e., the current iteration depends on some values computed in previous iterations).

  The induction variables in the authentication loop are #pwd#, #utmp#, #lusername#, #argc#, and #invalid#. Although an authentication routine should not have state variables, careful examination of the loop shows that login is correct. Section 9.7 discusses this issue further.

9.5.2. Program Slicing Needed

We have delayed the slicing for #ttyn# and #pwd# because their values are used in the slices for process creation, protection state, and authentication. The following are their slices. The slice for #ttyn# is very simple; #ttyname# is a library call. The slice for #pwd# in #main# is quite simple, but #doremotelogin# and #getloginname# change the value of #pwd#; therefore, we also slice these two procedures. #lusername# in #doremotelogin# is affected by #getstr#; therefore, we also examine #getstr#.

Slicing for #ttyn# in #main()#

```
103     char *ttyn;
172     ttyn = ttyname(0);
173     if (ttyn == (char *)0 || *ttyn == '\0')
174             ttyn = "/dev/tty??";
```

Slicing for #pwd# in #main()#

```
57      struct  passwd nouser = {"", "nope", -1, -1, -1, "", "", "", "" };
128                     usererr = doremotelogin(argv[2]);
186                     SCPYN(utmp.ut_name, "");
191                             SCPYN(utmp.ut_name, argv[1]);
199                             SCPYN(utmp.ut_name, lusername);
201                     getloginname(&utmp);
270                     if (*pwd->pw_shell == '\0')
271                             pwd->pw_shell = "/bin/sh";
272                     if (chdir(pwd->pw_dir) < 0) {
273                             if (chdir("/") < 0) {
276                             } else {
```

```
                279                                     pwd->pw_dir = "/";
                280                             }
                281                     }

Slicing for #pwd#, #lusername# in #getloginname()#
                409             while (up->ut_name[0] == ’\0’) {
                410                     namep = up->ut_name;
                412                     while ((c = getchar()) != ’0) {
                413                             if (c == ’ ’)
                414                                     c = ’_’;
                415                             if (c == EOF)
                416                                     exit(0);
                417                             if (namep < up->ut_name+NMAX)
                418                                     *namep++ = c;
                421             strncpy(lusername, up->ut_name, NMAX);
                422             lusername[NMAX] = 0;
                423             if ((pwd = getpwnam(lusername)) == NULL)
                424                     pwd = &nouser;

Slicing for #pwd#, #lusername# in #doremotelogin()#
                482     doremotelogin(host)
                483             char *host;
                484     {
                486             getstr(lusername, sizeof (lusername), "locuser");
                488             if (getuid()) {
                489                     pwd = &nouser;
                492             pwd = getpwnam(lusername);
                493             if (pwd == NULL) {
                494                     pwd = &nouser;

Slicing for #lusername# in #getstr()#
                500     getstr(buf, cnt, err)
                501             char *buf;
                502             int cnt;
                503             char *err;
                504     {
                505             char c;
                506
                507             do {
                508                     if (read(0, &c, 1) != 1)
                509                             exit(1);
                510                     if (--cnt < 0) {
                511                             printf("%s too long\r\n", err);
                512                             exit(1);
```

```
513                 }
514                 *buf++ = c;
515         } while (c != 0);
516     }
```

After the slices are determined, we can reexamine the process creation the protection states, the change of privilege, and the authentication tell-tale signs. The process #pwd->pw_shell# is created. The value is either the return value of #getpwnam(lusername)# at line 423 or #"/bin/sh"# at line 271. #getpwnam(user)# is a library function to return #user#'s password entry. Therefore, the process created is appropriate with respect to the login function. The protection state for the file #ttyn# is changed. The value of #ttyn# is either the return value of #ttyname(0)# at line 172 or #"/dev/tty??"# at line 174. #ttyname(0)# returns the terminal device associated with file descriptor 0. The login process changes the protection state of a terminal device if it has already obtained access; therefore, this change is not suspicious. The change of privilege occurs at lines 343 and 348 (see Section 9.5.1). They depend on the value of #pwd# defined at lines 423 or 424. The value from line 423 comes from #getpwnam()# library call, which should be fine. The value from line 424 is from #nouser# defined at line 57. The userid and groupid are both -1 and could cause security problems. We need to further examine the authentication code to see under what situations the userid and groupid will be set to -1. In the following, we show that #nouser# is not an acceptable password entry.

To slice the authentication loop, we use control-flow slicing to focus on the program fragment before and in the loop, and then we slice for the conditions (i.e., slicing for #invalid#) that the loop may exit. In login, the loop exits mean that the authentication is accepted. Since we slice for #invalid#, the error messages and the login-log statements are not shown. The slice is shown below.

Slicing for #invalid# in the authentication loop.

```
0       int     usererr = -1;
128                             usererr = doremotelogin(argv[2]);
182             invalid = FALSE;
183             do {
186                     SCPYN(utmp.ut_name, "");
191                             SCPYN(utmp.ut_name, argv[1]);
199                             SCPYN(utmp.ut_name, lusername);
201                             getloginname(&utmp);
202                             if (utmp.ut_name[0] == ’-’) {
204                                     invalid = TRUE;
205                                     continue;
206                             }
207                     }
208                     invalid = FALSE;
218                     if (usererr == -1 && *pwd->pw_passwd != ’ ’) {
```

```
222                         pp = getpass("Password:");
223                         namep = crypt(pp, pwd->pw_passwd);
225                         if (strcmp(namep, pwd->pw_passwd))
226                                 invalid = TRUE;
227                 }
242                 if (!invalid && pwd->pw_uid == 0 && !rootterm(tty)) {
250                         invalid = TRUE;
251                 }
270                 if (*pwd->pw_shell == ’  ’)
271                         pwd->pw_shell = "/bin/sh";
272                 if (chdir(pwd->pw_dir) < 0 && !invalid ) {
273                         if (chdir("/") < 0) {
275                                 invalid = TRUE;
276                         } else {
279                                 pwd->pw_dir = "/";
280                         }
281                 }
286                 if (!usererr && invalid)
287                         exit(1);
288         } while (invalid);
```

Slicing for #usererr# and #pwd# in #doremotelogin#.

```
482     doremotelogin(host)
483             char *host;
484     {
485             getstr(rusername, sizeof (rusername), "remuser");
486             getstr(lusername, sizeof (lusername), "locuser");
488             if (getuid()) {
489                     pwd = &nouser;
490                     return(-1);
491             }
492             pwd = getpwnam(lusername);
493             if (pwd == NULL) {
494                     pwd = &nouser;
495                     return(-1);
496             }
497             return(ruserok(host, (pwd->pw_uid == 0), rusername, lusername));
498     }
```

We see that the login name is #argv[1]#, #lusername#, or a string input in #getloginname()#. Then the password entry is obtained in #getloginname()#. If that user has no password entry, the default entry #nouser# defined at line 57 is used. The default encrypted password field, #"nope"#, cannot be matched with any password. #usererr#

initially has a value of -1 and will have other values if a remote login is successful (when #ruserok# at line 497 returns 0).

When the login is local (i.e., `usererr = -1`) and the encrypted password is not the empty string (i.e., `*pwd->pwd_passwd != ´\0´`), a password is read from the terminal, encrypted and compared with the password entry. #invalid# becomes true if the password does not match its entry.

Conclusively, we can say that a local login is granted only when the passwords match or the password entry is empty. We cannot say anything for remote logins because we also need to check the rlogin program and the remote login protocol before drawing a conclusion.

9.6. Another Example: Analysis of hangman.c

The game program hangman.c[5] is very simple in terms of slicing for any security-related properties because it does not write to files, create processes, access the network, change protection states, change privilege, have input-dependent system calls, or contain any authentication code. The source is in the Appendix.

hangman.c reads only one file: /usr/dict/words.

- 302 if ((Dict = fopen("/usr/dict/words", "r")) == 0) {

hangman.c uses the current time as the seed for the random number generator. The current time is obtained at statement 301 and used by #srand()#. After relating the flow with the static variable shared by the libraries #srand()# and #rand()#, we see that the time is used by #fseek()#. We conclude that there is no malicious time-dependent computation.

- 301 srand(time(0) + getpid());
  174 fseek(inf, abs(rand() % Dict_size), 0);

The hangman.c program has the following call structure. We further summarize the data used and generated by each procedure. No independent computation is found--data is passed as parameters, return values, and also as global variables.

Caller Callees

---

[5] The original hangman program consists of multiple files. We have combined them in order to run it through our tool.

main	setup playgame
getword	abs
endgame	prman prword prdata readch
getguess	readch
playgame	getword prword prdata prman getguess endgame

9.7. Discussion

Consider the usefulness of the program slicer with respect to the login program. On one hand, the analysis of the authentication shows that its logic is very confusing, both to the human and to the tool. Below we suggest a better way to structure the authentication logic. On the other hand, the program is well-behaved concerning strings and pointers.

The authentication loop handles two situations: local login and remote login. For local login, the value of #utmp.ut_name# is set at the first execution of the loop and remains unchanged. Variables like the #utmp# structure, whose sole purpose is to record a login entry into the log file, also remember the login userid. The login name appears in two variables at different places--#lusername# and #utmp.ut_name#--but they are used for the same purpose. #invalid# is a state variable used to distinguish the first iteration of the authentication loop from the rest.

The loop can be reorganized in a better way that minimizes the internal state and eliminates the spaghetti data dependence. Furthermore, parts of the program can be moved out of the authentication loop. Figure 9.1 is the revised authentication loop. It has one state variable, #rflag#, and always uses #utmp.ut_name# to remember the username. The revised loop is easier to understand and more sliceable than the original loop.

```
if (rflag) {
    /* remote login */
    SCPYN(utmp.ut_name, argv[1]); } else {
    /* local login */
    if (argc > 1)
        SCPYN(utmp.ut_name, argv[1]);
    else
        SCPYN(utmp.ut_name, ""); } do {
    invalid = FALSE;
    getloginname(&utmp);
    ...
    if (rflag)
        /* check whether remote user is ok */
        if (failed)
```

```
            rflag = 0; /* switch to local mode */
else
      /* get password and compare with the local password entry */
/* check whether logins are disabled */
/* for root logins, check for secure terminal */
...
/* if login is invalid, print message */
/* check whether we can change to that directory */
...  } while (invalid && ˜rflag)
```

With respect to well-behavedness, the original program is well written. Consider Fig. 9.2. #namep# at line 418 is easily verified to be bound between #up->ut_name# and #up->ut_name+NMAX#, where #NMAX# is the size of the #ut_name# string. Furthermore, at line 421, the program uses #strncpy()#, which limits the number of characters written to #lusername#. At line 422, the last character in #lusername# is assigned zero to ensure the string is null-terminated.

```
403  getloginname(up)
404        register struct utmp *up;
405  {
406        register char *namep;
407        char c;
408
409        while (up->ut_name[0] == ’  ’) {
410              namep = up->ut_name;
411              printf("login: ");
412              while ((c = getchar()) != ’0) {
413                    if (c == ’ ’)
414                          c = ’_’;
415                    if (c == EOF)
416                          exit(0);
417                    if (namep < up->ut_name+NMAX)
418                          *namep++ = c;
419              }
```

```
	}
	strncpy(lusername, up->ut_name, NMAX);
	lusername[NMAX] = 0;
	if ((pwd = getpwnam(lusername)) == NULL)
		pwd = &nouser;
}
```

CHAPTER 10

Conclusion and Future Directions

10. _10

In this chapter, we discuss issues related to automated malicious code detection. Section 10.1 reviews our approach, our goals and rationale, and our tool's advantages and limitations. We also discuss the completeness of malicious-code detection. Section 10.2 examines the formalization of maliciousness, discusses the issues in automated reasoning about programs, and describes the problems of applying flow analysis to reasoning about programs. Section 10.3 discusses the evaluation of our tool. Section 10.4 discusses the role of the MCF tool with respect to other security tools. Section 10.5 identifies other areas in which our tool will be useful. Section 10.6 describes extensions that could be made in the future.

10.1. Malicious Code Filter Approach

10.1.1. Summary

In this dissertation, we have addressed various issues in malicious code detection. We investigated and classified malicious code. We investigated how to distinguish malicious code from benign programs using the tell-tale sign and program-slicing approach. We determined how our approach can be defeated and invented countermeasures--the well-behavedness check. We generalized program slicing for malicious-code analysis. We applied and modified many static-analysis techniques to implement our proposed approach. One of them is the design and implementation of a very accurate data-dependence analyzer for the C language. We built a malicious-code filter with which further malicious-code research will be performed.

10.1.2. Goals and Rationale

The goal of our dissertation was to develop a method of detecting malicious code and vulnerabilities in system programs that may be exploited. After examining a wide range of defenses, including static methods and run-time techniques, we found out that run-time approaches limit and control the damages incurred by malicious code, but cannot expose malicious action before their activation. When it is desirable to eradicate rather than suppress malicious code, static analysis is the only viable solution.

It is not trivial to use static analysis to detect malicious code. Some techniques, such as program verification, can identify malicious properties, but their implementation cannot be automated in general. When it is, the problem domain is usually very limited. Some analyses, such as the syntactic analysis, can be solved automatically and efficiently in polynomial time, but the information collected is not strong enough to confirm that malicious code does not exist.

Our goal, which is very ambitious, is also generally unattainable: to develop an automatic method that can provide information strong enough to uncover malicious properties At present, no such methods exist. All program-verification systems are semiautomatic: the mechanical part is automated, and the rest is left to the user. For example, a verification-condition generator asks the user to supply loop invariants; a symbolic evaluator requires the user to decide which path to follow when a branch is reached. Poor systems ask the user to supply every detail; better systems automate the decision process and minimizes the interaction. The usefulness of such integrated human-machine systems depends mostly on how much effort is required from the user. We want to engineer our tool such that the user effort required to examine a program is minimized. We choose the following techniques to reduce the work of the user:

- Use tell-tale signs (so that the user does not need to specify the whole program).
- Use program slicing (so that the user does not need to examine the whole program).
- Select tell-tale signs that are suitable for slicing (so that the reduction of program size is significant).
- Provide a generic set of tell-tale signs (so that the same set can be applied to many different programs).
- Highlight suspected violations of well-behavedness (so that the user does not need to look through the program to search for ill-behavedness).

Finally, MCF is just a tool. The analyst selects the tell-tale signs of malicious behavior, uses MCF to slice and examine the program, and concludes whether the program is malicious or not.

10.1.3. Completeness

A tool is complete if it can detect malicious code in all programs. However, this is another unattainable goal. We relax our definition of the completeness to be the ability of the tool to solve problems in most situations and to determine the situations in problems it failed to solve. With security analysis, this determination of the situation is important because we do not want to draw any wrong (insecure) conclusions.

Heuristic malicious-code detection is incomplete by our definition because it detects certain instances of malicious code and ignores others. Any approach that examines only part of the source program is incomplete. Any run-time approach is incomplete because it cannot examine all possible execution sequences.

We believe our approach poses the minimal amount of work required for completeness. We argue that at a minimum, something like the well-behavedness checking must be contained in any complete approach. We have combined this checking with program slicing, which examines the minimum amount of code.

10.1.4. Advantages and Limitations

We made several major improvements over existing and proposed malicious-code detection methods. First, we do not require a formal specification of the program being analyzed. Second, our approach is general enough to handle classes of malicious code, whereas other approaches may handle only one instance of malicious code at a time. Third, our tool is programmable so that it can be adapted to handle new malicious code. Fourth, our tool is complete with respect to a specified property; previous work offering a similar level confidence does not exist.

Our tool is not without limitations. First, it cannot decide what is malicious. Second, it does not work with self-modifying programs (but can detect them). Third, the usefulness of our tool depends on how the program is written (i.e., the level of use of pointers, dynamic memory allocation, recursive data structures) and how well the program is written (i.e., spaghetti program, re-use of variables for different purposes). Fourth, the correctness of its result relies on the verification of well-behavedness property, which unfortunately cannot be completely automated. Fifth, our tool is unable to discriminate programmable programs (see Section 2.4.1) from specific malicious code.

10.2. Technical Difficulties

10.2.1. Formalization of Maliciousness

It is very difficult to define or formalize maliciousness. The same function (e.g., removing files from the current directory) performed at different times or by different programs can either be malicious or benign. The distinction lies in the intention of the user, and a program has no way to determine a user's intention! The method to solve the problem is to allow false positives (to identify a superset of maliciousness). We call this set suspicious. The human analyst will determine whether a suspicious property is a malicious one.

10.2.2. Programming Language Semantics

A problematic area in malicious-code detection is the semantics of the programming language. Our work analyzes C programs, which caused us a lot of problems. Some areas of C are obscure. Without a well-defined language, we cannot reason about the program. Therefore, we start with a better-defined intermediate language and translate C programs to the intermediate form.

Our prototype works with a subset of C that contains the more important features. Features not implemented can be simulated using implemented features. Therefore, programs in the full C language can be rewritten (or transformed) into the subset.

10.2.3. Flow Analysis

We use flow-analysis techniques to implement our filter. The most developed work on program analysis is the data-flow analysis. However, the focus is always on finding the most efficient algorithm to generate the data-flow information; therefore, most of the work has been restricted to some convenient program model. The most convenient model usually does not relate very well to any popular programming language. Furthermore, the difference in the goal of security analysis and program optimization is significant enough that we cannot directly adopt the optimization flow analysis to our tool.

The result is that we almost have to develop our flow analyzer from scratch based on previous work, including the design of an intermediate form, the C alias model, the pointer-mapping analysis, and the inter-procedural framework to solve the global flow problems.

10.2.4. Automated Reasoning

Automated program verification and theorem proving requires a very limited program model; programs or theorems that can be proved must be written in a very restricted way. Program size must be small because those algorithms are typically exponential. Special tailored heuristics, sometimes working with only particular examples, are introduced to cut down the exponential expansion.

When we verify a program, we complete two steps: generate the verification conditions from the program, and prove the verification conditions. The verification conditions capture the intentions of the program--what is being modified, and in what way. The generation of verification conditions is not automated except in very limited domain, such as the computation of toy numeric functions. The difficulty lies in understanding what the program is trying to achieve. The second difficulty is proving the theorems. In addition to perceiving what the program would accomplish, we need to demonstrate that our perception is correct. The state-of-the-art theorem provers cannot complete the job automatically. The Boyer-Moore theorem prover is somewhat automatic, but only works well under the supervision of an experienced user. Our naive attempt to prove a system of linear inequalities consisting of 12 equations that related 6 variables required 7 minutes of Sparc-2 CPU time to simplify the system to 1276 new cases, although some simple transformation could simplify the proof very much. HOL [HOL,Gordon] is a user-driven theorem prover. The user supplies tactics (i.e., proof directives) to direct the theorem prover. The creation of the tactics requires a high level of understanding of both the problem being proved and the underlying principle of the proof system. To prove a simple system requires a lot of expertise and time.

Our well-behavedness checker is a simple, automatic reasoning system with a home-grown verification-condition generator and a well-developed (Boyer-Moore) theorem prover. It is viable only because its domain is limited to integer-expressions in array accesses. Even then, we cannot establish a system that can prove all the array accesses. The VCG part only generates theorems expressible in a system of linear equations. In the worst case, the theorem generated still takes an exponential amount of time to be proved.

10.3. Evaluation

This dissertation aims to find out a way to analyze programs effectively and efficiently: the result of analysis should be useful in determining malicious code, and an analyst who uses our tool should attain a solution in less time than if he did the analysis himself.

Evaluating efficiency is difficult when we only have a prototype. Nevertheless, we tested our tell-tale signs on our malicious code[1] and some system vulnerabilities, and our tool located all of them. We also tested our tool on some benign programs, which were slightly modified BSD programs acceptable by our translator, and the slicing worked reasonably well.

We need to evaluate our tool on more malicious and benign programs of higher complexity to see how well our approach and tool works. To achieve this goal, our translator needs to accept the complete C language, and our flow analyzer must run faster in order to process large programs.

10.4. Role of Malicious Code Filter

Our tool examines a single source program before it is installed into the system. It does not guarantee that the installed programs are not modified, and it does not inhibit human intruders breaking into the system. Furthermore, it has no ability to enforce the proper setting of protection modes or limit the access rights of certain parties.

Therefore, our tool is complementary to many existing tools, such as DIDS, [dids,prototype] IDES, [ides,progress] and COPS, that address the human behavior and protection settings of the system. Our tool is also orthogonal to access-control mechanisms and dynamic analyzers. Access-control mechanisms deal with the relations between subjects and objects in a system, whereas our tool focuses on a single subject. A dynamic analyzer in a clean-room environment can analyze and dissect a malicious code; our tool does not provide this capability. In short, our tool does not solve all security problems.

10.5. Related Applications

The techniques developed in this dissertation are not limited to malicious-code detection. Our tool determines some fundamental properties of programs, and those properties will be useful in a variety of other areas, such as debugging, software reliability, and program test generation. Some related applications follow.

- **Static Analysis Experiment.** Other flow analyses should be easily programmed with our flow analyzer.
- **Dynamic Analysis.** The intermediate form could be interpreted with a dynamic tool. The data-flow information is useful in computing data-flow test coverage.

---

[1] These malicious code are realistic and based on known malicious code.

- **Analysis of Concurrent Programs.** Since our tool determined more accurate data dependence (i.e., with the analysis of pointer mappings), we can use it to prove the non-interference of concurrent modules.
- **Critical Applications.** Critical applications should have a well-behavedness check, e.g., static array-bounds checking. We can extend the concept of well-behavedness to include more features. Currently we only validate the correctness of the derived data-flow relations. There are other well-behavedness properties, such as run-time type conflicts, the release of memory that is still referenced, and the de-referencing of null pointers. These checks are useful in software-reliability analysis.
- **Software Engineering and Debugging.** Our tool determines procedure-call relations, data-flow relations between modules, pointer mappings, and possible values of variables. Other useful information, such as the lifetime analysis, may be obtained by flow analysis. The information derived with our tool can detect data flow anomalies: use of undefined variables, defined but unused variables, de-referencing of null pointers, and unreachable code. Slicing can reduce the program size for easier debugging.
- **Compiler Optimization.** Our tool can determine types for a dynamically typed language to generate better code. Furthermore, the tool can experiment with different compiler optimizations. Building a new optimization with an intermediate form is much easier than building it with a complete programming language.
- **Verification.** Flow analysis can precompute and derive certain information about the program, so that we use this information to simplify or speed up program verification. For example, we used flow analysis to validate array accesses in Chapter 8. The idea of a semantically well-defined intermediate form will also be useful for program verification.

10.6. Future Direction and Development

We foresee that programming languages will be designed with more concrete semantics and constructs that are easier to analyze. With high-assurance software, certain programming methodology and styles will be followed, leading to programs that are more sliceable and easier to understand.

In terms of the development of the Malicious Code Filter (MCF), we envision that MCF will be operated in two modes. In the first operating mode, MCF will act as a coarse filter, identifying those programs worthy of closer examination. A program will be analyzed and its properties summarized to allow the analyst to understand the possible effects of its execution. In its second mode of operation, MCF will support a more detailed examination of a sliced program, perhaps one that has been identified as such by the MCF. This analysis will investigate the exact nature of the previously identified suspicious property, determine its triggering conditions, and possibly discover additional suspicious properties. So far, MCF operates only under the first mode. Techniques such as symbolic evaluation, dynamic analysis, and testing will be useful.

- **Symbolic Evaluation.** Symbolic evaluation has two applications here: to further simplify the program slice to a more understandable form and to carry out symbolic execution. [Select,Elspas]
- **Dynamic Analysis.** Here the program is run in a clean-room environment, and its actions are observed to determine if they reflect malicious behavior. We propose to use the Dalek debugger [olsson,debugger] [Olsson,dataflow,approach,eventbased] to analyze programs parsed to the intermediate form. Particularly relevant to this work is the abstraction mechanism of Dalek; it can be programmed to flag sequences of events, which might indicate malicious activity.
- **Test-Case Generation.** Dalek requires the user to create test cases that will exercise the suspicious parts of the program. Cleverly hidden malicious code might require many test cases to be revealed, [Hamlet,testing,malicious] or the user might not create the particular test case(s) that is required. A systematic approach to test case generation will increase the likelihood of producing effective test cases. Of interest is the *coverage* of a set of test cases, that is, the fraction of the program exercised by the test cases. There are numerous measures of coverage, but the most relevant to the detection of malicious code appears to be data-flow coverage: test is created to exercise paths whose end points are the creation of a data object and its use. Other measures of coverage will be investigated.

APPENDIX A

11. _A

login.c

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
char copyright[] =
"@(#) Copyright (c) 1980 Regents of the University of California.\n\
 All rights reserved.\n";
#endif not lint

#ifndef lint
static char sccsid[] = "@(#)login.c     5.15 (Berkeley) 4/12/86";
#endif not lint

/*
 * login [ name ]
 * login -r hostname (for rlogind)
 * login -h hostname (for telnetd, etc.)
 */

#include <sys/param.h>
#include <sys/quota.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/file.h>

#include <sgtty.h>
#include <utmp.h>
#include <signal.h>
#include <pwd.h>
#include <stdio.h>
#include <lastlog.h>
```

```
#include <errno.h>
#include <ttyent.h>
#include <syslog.h>
#include <grp.h>

#define TTYGRPNAME      "tty"           /* name of group to own ttys */
#define TTYGID(gid)     tty_gid(gid)    /* gid that owns all ttys */

#define SCMPN(a, b)     strncmp(a, b, sizeof(a))
#define SCPYN(a, b)     strncpy(a, b, sizeof(a))

#define NMAX    sizeof(utmp.ut_name)
#define HMAX    sizeof(utmp.ut_host)

#define FALSE   0
#define TRUE    -1

char    nolog[] =       "/etc/nologin";
char    qlog[]  =       ".hushlogin";
char    maildir[30] =   "/usr/spool/mail/";
char    lastlog[] =     "/usr/adm/lastlog";
struct  passwd nouser = {"", "nope", -1, -1, -1, "", "", "", "" };
struct  sgttyb ttyb;
struct  utmp utmp;
char    minusnam[16] = "-";
char    *envinit[] = { 0 };             /* now set by setenv calls */
/*
 * This bounds the time given to login.  We initialize it here
 * so it can be patched on machines where it's too small.
 */
int     timeout = 60;

char    term[64];

struct  passwd *pwd;
char    *strcat(), *rindex(), *index(), *malloc(), *realloc();
int     timedout();
char    *ttyname();
char    *crypt();
char    *getpass();
char    *stypeof();
extern  char **environ;
extern  int errno;

struct  tchars tc = {
        CINTR, CQUIT, CSTART, CSTOP, CEOT, CBRK
};
struct  ltchars ltc = {
```

```
	CSUSP, CDSUSP, CRPRNT, CFLUSH, CWERASE, CLNEXT
};

struct winsize win = { 0, 0, 0, 0 };

int	rflag;
int	usererr = -1;
char	rusername[NMAX+1], lusername[NMAX+1];
char	rpassword[NMAX+1];
char	name[NMAX+1];
char	*rhost;

main(argc, argv)
	char *argv[];
{
	register char *namep;
	int pflag = 0, hflag = 0, t, f, c;
	int invalid, quietlog;
	FILE *nlfd;
	char *ttyn, *tty;
	int ldisc = 0, zero = 0, i;
	char **envnew;

	signal(SIGALRM, timedout);
	alarm(timeout);
	signal(SIGQUIT, SIG_IGN);
	signal(SIGINT, SIG_IGN);
	setpriority(PRIO_PROCESS, 0, 0);
	quota(Q_SETUID, 0, 0, 0);
	/*
	 * -p is used by getty to tell login not to destroy the environment
	 * -r is used by rlogind to cause the autologin protocol;
	 * -h is used by other servers to pass the name of the
	 * remote host to login so that it may be placed in utmp and wtmp
	 */
	while (argc > 1) {
		if (strcmp(argv[1], "-r") == 0) {
			if (rflag || hflag) {
				printf("Only one of -r and -h allowed\n");
				exit(1);
			}
			if (argv[2] == 0)
				exit(1);
			rflag = 1;
			usererr = doremotelogin(argv[2]);
			SCPYN(utmp.ut_host, argv[2]);
			argc -= 2;
			argv += 2;
```

```
			continue;
		}
		if (strcmp(argv[1], "-h") == 0 && getuid() == 0) {
			if (rflag || hflag) {
				printf("Only one of -r and -h allowed\n");
				exit(1);
			}
			hflag = 1;
			SCPYN(utmp.ut_host, argv[2]);
			argc -= 2;
			argv += 2;
			continue;
		}
		if (strcmp(argv[1], "-p") == 0) {
			argc--;
			argv++;
			pflag = 1;
			continue;
		}
		break;
	}
	ioctl(0, TIOCLSET, &zero);
	ioctl(0, TIOCNXCL, 0);
	ioctl(0, FIONBIO, &zero);
	ioctl(0, FIOASYNC, &zero);
	ioctl(0, TIOCGETP, &ttyb);
	/*
	 * If talking to an rlogin process,
	 * propagate the terminal type and
	 * baud rate across the network.
	 */
	if (rflag)
		doremoteterm(term, &ttyb);
	ttyb.sg_erase = CERASE;
	ttyb.sg_kill = CKILL;
	ioctl(0, TIOCSLTC, &ltc);
	ioctl(0, TIOCSETC, &tc);
	ioctl(0, TIOCSETP, &ttyb);
	for (t = getdtablesize(); t > 2; t--)
		close(t);
	ttyn = ttyname(0);
	if (ttyn == (char *)0 || *ttyn == '\0')
		ttyn = "/dev/tty??";
	tty = rindex(ttyn, '/');
	if (tty == NULL)
		tty = ttyn;
	else
		tty++;
```

```
    openlog("login", LOG_ODELAY, LOG_AUTH);
    t = 0;
    invalid = FALSE;
    do {
        ldisc = 0;
        ioctl(0, TIOCSETD, &ldisc);
        SCPYN(utmp.ut_name, "");
        /*
         * Name specified, take it.
         */
        if (argc > 1) {
            SCPYN(utmp.ut_name, argv[1]);
            argc = 0;
        }
        /*
         * If remote login take given name,
         * otherwise prompt user for something.
         */
        if (rflag && !invalid)
            SCPYN(utmp.ut_name, lusername);
        else {
            getloginname(&utmp);
            if (utmp.ut_name[0] == '-') {
                puts("login names may not start with '-'.");
                invalid = TRUE;
                continue;
            }
        }
        invalid = FALSE;
        if (!strcmp(pwd->pw_shell, "/bin/csh")) {
            ldisc = NTTYDISC;
            ioctl(0, TIOCSETD, &ldisc);
        }
        /*
         * If no remote login authentication and
         * a password exists for this user, prompt
         * for one and verify it.
         */
        if (usererr == -1 && *pwd->pw_passwd != '\0') {
            char *pp;

            setpriority(PRIO_PROCESS, 0, -4);
            pp = getpass("Password:");
            namep = crypt(pp, pwd->pw_passwd);
            setpriority(PRIO_PROCESS, 0, 0);
            if (strcmp(namep, pwd->pw_passwd))
                invalid = TRUE;
        }
```

```
	/*
	 * If user not super-user, check for logins disabled.
	 */
	if (pwd->pw_uid != 0 && (nlfd = fopen(nolog, "r")) > 0) {
		while ((c = getc(nlfd)) != EOF)
			putchar(c);
		fflush(stdout);
		sleep(5);
		exit(0);
	}
	/*
	 * If valid so far and root is logging in,
	 * see if root logins on this terminal are permitted.
	 */
	if (!invalid && pwd->pw_uid == 0 && !rootterm(tty)) {
		if (utmp.ut_host[0])
			syslog(LOG_CRIT,
			    "ROOT LOGIN REFUSED ON %s FROM %.*s",
			    tty, HMAX, utmp.ut_host);
		else
			syslog(LOG_CRIT,
			    "ROOT LOGIN REFUSED ON %s", tty);
		invalid = TRUE;
	}
	if (invalid) {
		printf("Login incorrect\n");
		if (++t >= 5) {
			if (utmp.ut_host[0])
				syslog(LOG_CRIT,
				    "REPEATED LOGIN FAILURES ON %s FROM %.*s, %.*s",
				    tty, HMAX, utmp.ut_host,
				    NMAX, utmp.ut_name);
			else
				syslog(LOG_CRIT,
				    "REPEATED LOGIN FAILURES ON %s, %.*s",
					tty, NMAX, utmp.ut_name);
			ioctl(0, TIOCHPCL, (struct sgttyb *) 0);
			close(0), close(1), close(2);
			sleep(10);
			exit(1);
		}
	}
	if (*pwd->pw_shell == '\0')
		pwd->pw_shell = "/bin/sh";
	if (chdir(pwd->pw_dir) < 0 && !invalid ) {
		if (chdir("/") < 0) {
			printf("No directory!\n");
			invalid = TRUE;
```

```
				} else {
					printf("No directory! %s\n",
					    "Logging in with home=/");
					pwd->pw_dir = "/";
				}
			}
			/*
			 * Remote login invalid must have been because
			 * of a restriction of some sort, no extra chances.
			 */
			if (!usererr && invalid)
				exit(1);
		} while (invalid);
/* committed to login turn off timeout */
		alarm(0);

		if (quota(Q_SETUID, pwd->pw_uid, 0, 0) < 0 && errno != EINVAL) {
			if (errno == EUSERS)
				printf("%s.\n%s.\n",
				    "Too many users logged on already",
				    "Try again later");
			else if (errno == EPROCLIM)
				printf("You have too many processes running.\n");
			else
				perror("quota (Q_SETUID)");
			sleep(5);
			exit(0);
		}
		time(&utmp.ut_time);
		t = ttyslot();
		if (t > 0 && (f = open("/etc/utmp", O_WRONLY)) >= 0) {
			lseek(f, (long)(t*sizeof(utmp)), 0);
			SCPYN(utmp.ut_line, tty);
			write(f, (char *)&utmp, sizeof(utmp));
			close(f);
		}
		if ((f = open("/usr/adm/wtmp", O_WRONLY|O_APPEND)) >= 0) {
			write(f, (char *)&utmp, sizeof(utmp));
			close(f);
		}
		quietlog = access(qlog, F_OK) == 0;
		if ((f = open(lastlog, O_RDWR)) >= 0) {
			struct lastlog ll;

			lseek(f, (long)pwd->pw_uid * sizeof (struct lastlog), 0);
			if (read(f, (char *) &ll, sizeof ll) == sizeof ll &&
			    ll.ll_time != 0 && !quietlog) {
				printf("Last login: %.*s ",
```

```
                            24-5, (char *)ctime(&ll.ll_time));
                        if (*ll.ll_host != '\0')
                                printf("from %.*s\n",
                                    sizeof (ll.ll_host), ll.ll_host);
                        else
                                printf("on %.*s\n",
                                    sizeof (ll.ll_line), ll.ll_line);
                }
                lseek(f, (long)pwd->pw_uid * sizeof (struct lastlog), 0);
                time(&ll.ll_time);
                SCPYN(ll.ll_line, tty);
                SCPYN(ll.ll_host, utmp.ut_host);
                write(f, (char *) &ll, sizeof ll);
                close(f);
        }
        chown(ttyn, pwd->pw_uid, TTYGID(pwd->pw_gid));
        if (!hflag && !rflag)                                   /* XXX */
                ioctl(0, TIOCSWINSZ, &win);
        chmod(ttyn, 0620);
        setgid(pwd->pw_gid);
        strncpy(name, utmp.ut_name, NMAX);
        name[NMAX] = '\0';
        initgroups(name, pwd->pw_gid);
        quota(Q_DOWARN, pwd->pw_uid, (dev_t)-1, 0);
        setuid(pwd->pw_uid);
        /* destroy environment unless user has asked to preserve it */
        if (!pflag)
                environ = envinit;

        /* set up environment, this time without destruction */
        /* copy the environment before setenving */
        i = 0;
        while (environ[i] != NULL)
                i++;
        envnew = (char **) malloc(sizeof (char *) * (i + 1));
        for (; i >= 0; i--)
                envnew[i] = environ[i];
        environ = envnew;

        setenv("HOME=", pwd->pw_dir, 1);
        setenv("SHELL=", pwd->pw_shell, 1);
        if (term[0] == '\0')
                strncpy(term, stypeof(tty), sizeof(term));
        setenv("TERM=", term, 0);
        setenv("USER=", pwd->pw_name, 1);
        setenv("PATH=", ":/usr/ucb:/bin:/usr/bin", 0);

        if ((namep = rindex(pwd->pw_shell, '/')) == NULL)
```

```
		namep = pwd->pw_shell;
	else
		namep++;
	strcat(minusnam, namep);
	if (tty[sizeof("tty")-1] == ’d’)
		syslog(LOG_INFO, "DIALUP %s, %s", tty, pwd->pw_name);
	if (pwd->pw_uid == 0)
		if (utmp.ut_host[0])
			syslog(LOG_NOTICE, "ROOT LOGIN %s FROM %.*s",
			    tty, HMAX, utmp.ut_host);
		else
			syslog(LOG_NOTICE, "ROOT LOGIN %s", tty);
	if (!quietlog) {
		struct stat st;

		showmotd();
		strcat(maildir, pwd->pw_name);
		if (stat(maildir, &st) == 0 && st.st_size != 0)
			printf("You have %smail.\n",
				(st.st_mtime > st.st_atime) ? "new " : "");
	}
	signal(SIGALRM, SIG_DFL);
	signal(SIGQUIT, SIG_DFL);
	signal(SIGINT, SIG_DFL);
	signal(SIGTSTP, SIG_IGN);
	execlp(pwd->pw_shell, minusnam, 0);
	perror(pwd->pw_shell);
	printf("No shell\n");
	exit(0);
}

getloginname(up)
	register struct utmp *up;
{
	register char *namep;
	char c;

	while (up->ut_name[0] == ’\0’) {
		namep = up->ut_name;
		printf("login: ");
		while ((c = getchar()) != ’\n’) {
			if (c == ’ ’)
				c = ’_’;
			if (c == EOF)
				exit(0);
			if (namep < up->ut_name+NMAX)
				*namep++ = c;
		}
```

```
	}
	strncpy(lusername, up->ut_name, NMAX);
	lusername[NMAX] = 0;
	if ((pwd = getpwnam(lusername)) == NULL)
		pwd = &nouser;
}

timedout()
{

	printf("Login timed out after %d seconds\n", timeout);
	exit(0);
}

int	stopmotd;
catch()
{

	signal(SIGINT, SIG_IGN);
	stopmotd++;
}

rootterm(tty)
	char *tty;
{
	register struct ttyent *t;

	if ((t = getttynam(tty)) != NULL) {
		if (t->ty_status & TTY_SECURE)
			return (1);
	}
	return (0);
}

showmotd()
{
	FILE *mf;
	register c;

	signal(SIGINT, catch);
	if ((mf = fopen("/etc/motd", "r")) != NULL) {
		while ((c = getc(mf)) != EOF && stopmotd == 0)
			putchar(c);
		fclose(mf);
	}
	signal(SIGINT, SIG_IGN);
}

```

```
#undef  UNKNOWN
#define UNKNOWN "su"

char *
stypeof(ttyid)
	char *ttyid;
{
	register struct ttyent *t;

	if (ttyid == NULL || (t = getttynam(ttyid)) == NULL)
		return (UNKNOWN);
	return (t->ty_type);
}

doremotelogin(host)
	char *host;
{
	getstr(rusername, sizeof (rusername), "remuser");
	getstr(lusername, sizeof (lusername), "locuser");
	getstr(term, sizeof(term), "Terminal type");
	if (getuid()) {
		pwd = &nouser;
		return(-1);
	}
	pwd = getpwnam(lusername);
	if (pwd == NULL) {
		pwd = &nouser;
		return(-1);
	}
	return(ruserok(host, (pwd->pw_uid == 0), rusername, lusername));
}

getstr(buf, cnt, err)
	char *buf;
	int cnt;
	char *err;
{
	char c;

	do {
		if (read(0, &c, 1) != 1)
			exit(1);
		if (--cnt < 0) {
			printf("%s too long\r\n", err);
			exit(1);
		}
		*buf++ = c;
	} while (c != 0);
```

```
}

char	*speeds[] =
    { "0", "50", "75", "110", "134", "150", "200", "300",
      "600", "1200", "1800", "2400", "4800", "9600", "19200", "38400" };
#define NSPEEDS (sizeof (speeds) / sizeof (speeds[0]))

doremoteterm(term, tp)
	char *term;
	struct sgttyb *tp;
{
	register char *cp = index(term, '/'), **cpp;
	char *speed;

	if (cp) {
		*cp++ = '\0';
		speed = cp;
		cp = index(speed, '/');
		if (cp)
			*cp++ = '\0';
		for (cpp = speeds; cpp < &speeds[NSPEEDS]; cpp++)
			if (strcmp(*cpp, speed) == 0) {
				tp->sg_ispeed = tp->sg_ospeed = cpp-speeds;
				break;
			}
	}
	tp->sg_flags = ECHO|CRMOD|ANYP|XTABS;
}

/*
 * Set the value of var to be arg in the Unix 4.2 BSD environment env.
 * Var should end with '='.
 * (bindings are of the form "var=value")
 * This procedure assumes the memory for the first level of environ
 * was allocated using malloc.
 */
setenv(var, value, clobber)
	char *var, *value;
{
	extern char **environ;
	int index = 0;
	int varlen = strlen(var);
	int vallen = strlen(value);

	for (index = 0; environ[index] != NULL; index++) {
		if (strncmp(environ[index], var, varlen) == 0) {
			/* found it */
			if (!clobber)
```

```
				return;
			environ[index] = malloc(varlen + vallen + 1);
			strcpy(environ[index], var);
			strcat(environ[index], value);
			return;
		}
	}
	environ = (char **) realloc(environ, sizeof (char *) * (index + 2));
	if (environ == NULL) {
		fprintf(stderr, "login: malloc out of memory\n");
		exit(1);
	}
	environ[index] = malloc(varlen + vallen + 1);
	strcpy(environ[index], var);
	strcat(environ[index], value);
	environ[++index] = NULL;
}

tty_gid(default_gid)
	int default_gid;
{
	struct group *getgrnam(), *gr;
	int gid = default_gid;

	gr = getgrnam(TTYGRPNAME);
	if (gr != (struct group *) 0)
		gid = gr->gr_gid;

	endgrent();

	return (gid);
}
```

hangman.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>



# define        MINLEN  6
# define        MAXERRS 7
# define        BUFSIZ 1024
# define        DICT    "/usr/dict/words"


struct ERR_POS {
        int     y;
        int x;
        char    ch;
} Err_pos[7] = {
        {  2, 10, ’O’ },
        {  3, 10, ’|’ },
        {  4, 10, ’|’ },
        {  5,  9, ’/’ },
        {  3,  9, ’/’ },
        {  3, 11, ’\\’ },
        {  5, 11, ’\\’ }
};

struct  stat
{
        short   st_dev;
        short   st_ino;
        short st_mode;
        short   st_nlink;
        short   st_uid;
        short   st_gid;
        short   st_rdev;
        int     st_size;
        int     st_atime;
        int     st_spare1;
        int     st_mtime;
        int     st_spare2;
        int     st_ctime;
        int     st_spare3;
        long    st_blksize;
        long    st_blocks;
        long    st_spare4[2];
```

```
} sbuf;

int     Guessed[26];

char    Word[1024],
        Known[1024],
        *Noose_pict[] = {
                "     _______",
                "     |    |",
                "     |",
                "     |",
                "     |",
                "     |",
                "   __|______",
                "   |     |___",
                "   |_________|",
                0
        };

int     Errors,
        Wordnum = 0;

float   Average = 0.0;

FILE    *Dict = 0;

int     Dict_size;


main()
{
        setup();
        for (;;) {
                Wordnum++;
                playgame();
                Average = (Average * (Wordnum - 1) + Errors) / Wordnum;
        }
}

endgame()
{
        register char   ch;

        prman();
        if (Errors >= 7)
                Errors = 7 + 2;
        prword();
        prdata();
```

```
	if (Errors > 7)
		printf("Sorry, the word was \"%s\"\n", Word);
	else
		printf("You got it!\n");

	for (;;) {
		printf("Another word? ");
		if ((ch = readch()) == ’n’)
			exit();
		else if (ch == ’y’)
			break;
		printf("Please type ’y’ or ’n’");
	}
}


getguess()
{
	register int	i;
	register int	ch;
	register int	correct;

	printf("Guess: ");
	for (;;) {
		ch = readch();
		if (isalpha(ch)) {
			if (isupper(ch))
				ch = tolower(ch);
			if (Guessed[ch - ’a’])
				printf("Already guessed ’%c’\n", ch);
			else
				break;
		}
		else if (ch == 4)
			exit();
		else if (ch != ’\n’)
			printf("\nNot a valid guess: ’%c’\n",ch);
	}
	Guessed[ch - ’a’] = 1;
	correct = 0;
	for (i = 0; Word[i] != ’\0’; i++)
		if (Word[i] == ch) {
			Known[i] = ch;
			correct = 1;
		}
	if (!correct)
		Errors++;
}
```

```

readch()
{
        int     cnt, r;
        char    ch;

        cnt = 0;
        for (;;) {
                if (read(0, &ch, sizeof ch) <= 0)
                {
                        if (++cnt > 100)
                                exit();
                }
                else
                        return ch;
        }
}

/*
 * getword:
 *      Get a valid word out of the Dictionary file
 */
getword()
{
        FILE            *inf;
        char            *wp, *gp;
        int cont;

        inf = "/usr/dict/words";
        while (cont) {
            cont = 0;
            fseek(inf, abs(rand() % Dict_size), 0);
            if (fgets(Word, 1024, inf) != 0)
                if (fgets(Word, 1024, inf) != 0) {
                    Word[strlen(Word) - 1] = '\0';
                    if (strlen(Word) > 6)
                        for (wp = Word; *wp; wp++)
                            if (!islower(*wp))
                                cont =1 ;
                }
        }
        gp = Known;
        wp = Word;
        while (*wp) {
                *gp = '-';
                gp++;
                wp++;
        }
```

```
        *gp = ’\0’;
}

/*
 * abs:
 *      Return the absolute value of an integer
 */
abs(i)
int    i;
{
    if (i < 0)
        return  -i;
    else
        return i;
}

/*
 * playgame:
 *      play a game
 */
playgame()
{
        register int    *bp;

        getword();
        Errors = 0;
        bp = Guessed;
        while (bp < &Guessed[26]) {
                *bp = 0;
                bp++;
            }
        while (Errors < 7 && index(Known, ’-’) != 0) {
                prword();
                prdata();
                prman();
                getguess();
        }
        endgame();
}

/*
 * prdata:
 *      Print out the current guesses
 */
prdata()
{
        int     *bp;

```

```
        printf("Guessed: ");
        bp = Guessed;
        while (bp < &Guessed[26])
                if (*bp++)
                        putchar((bp - Guessed) + 'a' - 1);
        putchar('\n');
        printf("Word #:           %d\n", Wordnum);
        printf("Current Average:   %.3f\n",
                        (Average * (Wordnum - 1) + Errors) / Wordnum);
        printf("Overall Average:   %.3f\n", Average);
}

/*
 * prman:
 *      Print out the man appropriately for the give number
 *      of incorrect guesses.
 */
prman()
{
        int     i;
        char line[9][100];
        char **sp;

        i = 0;
        for (sp = Noose_pict; *sp != 0; sp++) {
            strcpy(line[i], *sp);
            strcat(line[i], "              ");
            i++;
        }

        for (i = 0; i < Errors; i++)
            line[Err_pos[i].y][Err_pos[i].x] =  Err_pos[i].ch;

        for (i = 0; i < 9; i++) {
             printf(line[i]);
             putchar('\n');
        }

}

/*
 * prword:
 *      Print out the current state of the word
 */
prword()
{
        printf("Known: %s\n", Known);
}
```

```

/*
 * setup:
 *      Set up the strings on the screen.
 */
setup()
{
        register char           **sp;

        for (sp = Noose_pict; *sp != 0; sp++) {
            printf(*sp);
            putchar('\n');
        }

        srand(time(0) + getpid());
        if ((Dict = fopen("/usr/dict/words", "r")) == 0) {
                perror("/usr/dict/words");
                exit(1);
        }
        fstat(fileno(Dict), &sbuf);
        Dict_size = sbuf.st_size;
}
```