DYNAMIC INFORMATION FLOW ANALYSIS, SLICING AND PROFILING

by

WASSIM A. MASRI

Submitted in partial fulfillment of the requirements
For the degree of Doctor of Philosophy

Dissertation Advisor: Dr. Andy Podgurski

Electrical Engineering and Computer Science Department

CASE WESTERN RESERVE UNIVERSITY

January 2005
CASE WESTERN RESERVE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

We hereby approve the dissertation of

candidate for the Ph.D. degree*

______________________________________________________
Dr. Andy Podgurski

______________________________________________________
Dr. Lee White

______________________________________________________
Dr. Vincenzo Liberatore

______________________________________________________
Dr. Jiayang Sun

*We also certify that written approval has been obtained for any proprietary material contained therein.
Chapter 7: Case Studies

7.2 Case Study 1: JConsole ................................................................. 50
7.3 Case Study 2: MeetingScheduler ................................................... 51
7.4 Case Study 3: DefaultServlet ......................................................... 54
7.5 Case study 4: Path disclosure ......................................................... 55
7.6 Case study 5: EmploymentVerifier .................................................. 57
7.7 Time and storage impact .............................................................. 59

Chapter 8: Implicit information flows .................................................. 62
8.1 if-else statements ........................................................................... 63
8.2 if statements .................................................................................... 66
8.3 Other control statements ............................................................... 68
8.4 Transformation algorithm ............................................................. 71
8.5 Implicit flows with non-simple targets ......................................... 75
8.6 Inter-procedural implicit flows ....................................................... 78
8.7 Arrays and array elements ............................................................. 81

Chapter 9: Test-case filtering ............................................................. 86
9.1 Coverage-based techniques ......................................................... 87
9.2 OBT techniques ............................................................................ 87
9.3 Subject Programs ......................................................................... 88
9.4 Basic coverage maximization results and discussions ..................... 91
9.5 OBT results and discussions ......................................................... 97

Chapter 10: Strength and length of information flows ............................. 104
10.1 Strength ....................................................................................... 104
10.2 Length ......................................................................................... 105
10.3 Strength vs. length ....................................................................... 106

Chapter 11: Application-Based Intrusion Detection .................................. 112
11.1 Misuse detection ......................................................................... 113
11.2 Anomaly detection ...................................................................... 114

Chapter 12: Future work ..................................................................... 119

Chapter 13: Conclusions ................................................................... 120

Chapter 14: Appendix A ................................................................. 122
14.1 Instrumenter ............................................................................... 122
14.2 Profiler: Data Structures ............................................................. 124
14.3 Profiler: Methods’ Implementations ............................................ 126
List of Tables

1. \textit{ipd}(v), \textit{CD}(v) and \textit{DCD}(v) relationships of the CFG in Figure 7.......................... 25
2. Sample trace in the CFG of Figure 7 .................................................................26
3. \textit{DynSlice} and \textit{InfoFlow} for the program in Figure 10 ................................. 49
4. Time and storage impact for various programs ....................................................61
5. Unique execution counts for the various types of profiles ..............................90
6. Results for conducting basic coverage maximization ..................................92
# List of Figures

1. Java code that might leak sensitive information ........................................3
2. High-level architecture of the debugging process .................................6
3. High-level architecture of the process enforcing information flow policies .......7
4. Java code with a defect that causes an insecure information flow ..................9
5. High-level architecture of the observation-based testing process ...............11
6. jRapture high-level architecture ................................................................20
7. Control flow graph for an unstructured function ......................................25
8. Direct Dynamic Control Dependence Algorithm .....................................32
9. InfoFlow and DynSlice algorithm .......................................................42
10. Java byte code and its java-like equivalent .........................................48
11. Insecure Java code implementing the meeting scheduler .......................53
12. Insecure servlet that might disclose the root path of the webserver .............57
13. Insecure Java code implementing the employment verifier ......................59
14. Original and transformed code ..........................................................64
15. Original and transformed code ..........................................................66
16. Original and transformed Java code involving various control statements ......69
17. Original and transformed Java code involving various control statements ......70
18. Transformation algorithm ....................................................................72
19. Original and transformed code with and without adding vertices ..............73
20. Control flow graph of the code in Figure 19-a and 19-b .........................73
21. Control flow graph of the code in Figure 19-c ......................................73
22. Original and transformed code .............................................................77
23. Original and transformed code ............................................................79
24. Class hierarchy, original code and transformed code .........................80
25. Original code, transformed code, explicit flow, explicit and implicit flow ....83
26. Transformed code based on array region analysis ..................................84
27. Transformed code where the targets involve array elements referencing objects.85
28. Basic coverage maximization and random sampling for Javac ..................94
29. Basic coverage maximization and random sampling for Xerces ...............95
30. Basic coverage maximization and random sampling for JTidy ...................96
31. One-per-cluster sampling and random sampling for javac .....................98
32. One-per-cluster sampling and random sampling for Xerces ...................99
33. One-per-cluster sampling and random sampling for JTidy .....................100
34. Failure-pursuit sampling and random sampling for javac ....................101
35. Failure-pursuit sampling and random sampling for Xerces ...................102
36. Failure-pursuit sampling and random sampling for JTidy .....................103
37. Length vs. average correlation coefficient for Xerces ...........................109
38. Length vs. average correlation coefficient for Xerces ...........................110
39. Length vs. strength data for two JTidy executions and one Diff execution .....111
40. MDS plot of the Tomcat executions ..................................................118
41. Class diagram classifying the various Java byte code instructions ..........123
42. Class diagram classifying the various DataBags .................................132
43. Class diagram classifying the various variable designators .................133
**Glossary**

**Information flow:** Information flow occurs from object \( y \) (*source*) to object \( x \) (*target* or *sink*) whenever information stored in \( y \) is propagated directly or indirectly to object \( x \), i.e., if information about \( y \) can be inferred by examining \( x \).

**Information flow analysis:** Information flow analysis is concerned with monitoring and regulating the flow of information among objects throughout a system and between the system and the outside world.

**Information flow policy:** Information flow policies define the way information can securely move throughout a system and ultimately between the system and the outside world.

**Program Slicing:** Program Slicing is concerned with finding all statements in a program that directly or indirectly influence another statement in a program.

**Forward computing:** Forward computing slicing algorithms operate in tandem with program execution, they compute the slice of every executing statement and at any point during execution they have available the slices associated with every active object in the program.
Test-case filtering: Test-case filtering is concerned with selecting a subset of a test suite that is capable of covering most or all of the defects covered by the original test suite.
Dynamic Information Flow Analysis, Slicing and Profiling

Abstract

by

Wassim A. Masri

Dynamic information flow analysis is concerned with the runtime monitoring and regulation of the flow of information among objects throughout a system and ultimately between the system and the outside world. A new approach to dynamic information flow analysis is presented that can be used to detect and debug insecure flows in programs. It can be applied offline to validate and debug a program against an information flow policy, or, when fast response is not critical, it can be applied online to prevent illegal flows in deployed programs. Dynamic mechanisms are inherently unable to detect implicit information flows; our approach incorporates an optional static preprocessing phase that identifies implicit flows and transforms them into explicit ones. The resulting hybrid mechanism is therefore capable of detecting both explicit and implicit information flows.

Program slicing is an integral part of debugging against an information flow policy. Forward computing slicing algorithms, which do not require a previously stored execution trace, are especially suited for interactive debugging. This dissertation proposes a dynamic slicing algorithm, which is characterized as forward computing, precise and applicable to unstructured programs.
Observation-based testing (OBT) is an approach in which executions of a program are profiled and then analyzed using cluster analysis and sampling methods to identify unusual or suspicious executions for manual auditing. This technique can be used to detect ordinary failures as well as intrusive behaviors. It can also be the basis for test-case filtering. This dissertation contributes to observation-based testing research by devising two profiling techniques that capture a relatively higher level of detail from a program execution namely, information flow profiling and slice profiling. In order to empirically verify their relative efficiency, we used them as well as other types of profiles such as function calls and data flow profiles to conduct test-case filtering experiments. The test-case filtering experiments involved OBT techniques as well as coverage-based techniques. The comparative results are presented and discussed.

Finally, this dissertation presents a prototype tool for detecting and debugging insecure information flows in Java byte programs, it is also capable of generating information flow and slice execution profiles.
Chapter 1

Introduction

Information flows in programs can cause security violations if they leak confidential information or lead to the corruption of high integrity data. Such insecure information flows usually occur as a result of security attacks, but they can also occur during normal use. Also, ordinary failures, such as the ones caused by data corruption, can result from (secure) information flows. Therefore, information flow analysis can benefit both the security and reliability aspects of a software system. Information flow policies define the way information can securely move throughout a system and ultimately between the system and the outside world, they are designed and enforced to guard against insecure information flows. Dynamic information flow analysis or DIFA is concerned with the runtime monitoring and regulation of the information flow among objects throughout a system. This dissertation presents a new approach to DIFA that can be used to detect and debug insecure flows in programs.

Information flows can be either explicit or implicit. Our approach to DIFA is capable of detecting most implicit flows at runtime, in addition to explicit ones. Explicit flows can result from either data or control dependences between program statements. Implicit flows can result from control dependences only. For example, there is an explicit flow from a variable (or a return value of a function) used in an assignment statement to the variable defined by the statement. Given a conditional statement $p$, there is an information flow from a variable used in $p$ to a variable defined in the scope of $p$, this flow is explicit when $p$ evaluates to true (its scope is entered) and implicit when $p$ evaluates to false. For example, consider the statements
\[
x = 1; \text{if } y = 0 \text{ then } x = 0;
\]

where \( y \) is either 0 or 1. By observing the final value of \( x \), one can deduce the value of \( y \) whether or not the \text{then} clause is executed. The flow is explicit if the \text{then} clause is executed and implicit if it is not\(^1\). Since dynamic analysis considers only executing statements, it is inherently unable to detect implicit information flows as described above. Our approach incorporates an optional static preprocessing phase that identifies implicit flows and transforms them into explicit ones. The resulting hybrid mechanism is therefore capable of detecting most implicit flows at runtime, in addition to explicit ones. Note that our approach is the first to address the detection of implicit flows in the dynamic setting.

\( \text{DIFA} \) is closely related to \textit{dynamic slicing} \cite{[81][4]}, which is a debugging technique that seeks to identify at runtime the set of program statements, called a \textit{slice}, that could be responsible for an erroneous program state that occurred at a particular location in a program. This dissertation presents an algorithm for \( \text{DIFA} \) that detects insecure information flows and another algorithm for dynamic slicing that can be used for debugging insecure information flows as well as for conventional debugging. Both algorithms are \textit{precise, forward computing} and are applicable to structured and unstructured programs. A precise \( \text{DIFA} \) algorithm will generate fewer false positives and a precise slicing algorithm enables more effective debugging. Forward computing algorithms, which operate in tandem with program execution and do not require a previously stored execution trace, are especially suited for online analysis and interactive debugging. The precision of our algorithms, even in the presence of unstructured

\(^{1}\) Note that this characterization of implicit and explicit flows is somewhat different from that presented in \cite{[13]}.
void sellOrHold(double current, double purchase, double threshold)
{
    double profit;
    boolean action;

    1.       profit = 100*(current - purchase)/purchase;

    2.       if (profit >= threshold)
       {
            3.    action= true;
       }
    else
    {
        4.    action= false;
    }
    5.       submitTransaction(action);
}

Figure 1 – Java code that might leak sensitive information to the outside world.

constructs, stems from the fact that they are dynamic and are based on the graph-theoretic
definition of control dependence whose soundness has been theoretically proven [60].

We implemented our algorithms in a prototype tool for Java byte code programs.
Recognizing that security needs often change after deployment (e.g. what is considered
secure today might not be considered secure tomorrow or vice versa), our tool supports
configurable information flow policies, a feature that static and type checking systems
lack [67][13]. Given its dynamic nature, our tool easily handles language features that
are generally hard to handle in a static context, such as pointers, arrays and dynamic
binding. Our tool handles both intra-procedural and inter-procedural dependences and
also handles data flows between threads in multi-threaded programs. In the Java code
shown in Figure 1, our DIFA tool detects that at line 1, information flowed from the
objects \{\text{current, purchase}\}^2 \) to the variable \text{profit}. Also at line 3 (or line 4) it detects that a flow occurred from the objects \{\text{profit, threshold, current, purchase}\} to the variable \text{action}, and more importantly that a flow occurred from the objects \{\text{profit, threshold, current, purchase, action}\} to the outside world through the \text{submitTransaction} method. Additionally, if for example the \text{threshold} variable is known to be sensitive, our tool secures the execution of the above code by preventing line 5 from executing and disclosing the value of \text{action}. Otherwise, an outsider knowledgeable about the code might deduce information about \text{threshold} by observing the value of \text{action}.

This chapter presents a high-level architecture of our proposed approach (Section 1.1), demonstrates the benefits of dynamic slicing to \text{DIFA} (Section 1.2), argues the use of information flow and slice profiling in test-case filtering (Section 1.3), and finally gives a roadmap to the remainder of the document (Section 1.4).

1.1 Detecting and debugging insecure information flows

This dissertation proposes a new approach to detecting and debugging insecure information flows. This approach can be applied offline to validate and debug a program against an information flow policy, or, when fast response is not critical, it can be applied online to prevent illegal flows in deployed programs.

Figure 2 shows the high-level architecture of our validation process against an information flow policy. The original Java byte code program is instrumented to enable the monitoring of data and control flows. An information flow policy is defined and

---

^2 For simplicity, we are not considering the objects that \text{current, purchase or threshold} might depend on.
registered in our tool. The instrumented program is run using existing test suites or data captured in the field (see Section 2.4). When a policy is *about to be* violated, the executing thread will be suspended at the statement where the violation was to occur. The programmer can then examine the information flows and program slices of all active objects in addition to conventional debugging entities such as global symbols and local symbols in each stack frame. Code changes might follow and the cycle is repeated as necessary.

Figure 3 shows the high-level architecture of our process enforcing an information flow policy during deployment. The instrumented program is deployed live. When the registered information flow policy is *about to be* violated, the execution will stop and the tool will log the information flows and program slices at the violation site to be examined by a programmer or a security analyst later on. Note that our tool detects the violation right before it occurs, therefore the illegal flow does not actually take place. Also, given the overhead introduced by our tool, it is feasible to use this setup only when fast application response is not critical.
Figure 2 – High-level architecture of the debugging process against information flow policies
Figure 3 - High-level architecture of the process enforcing information flow policies during deployment.
1.2 Dynamic slicing

Dynamic slicing is concerned with finding, at runtime, all statements in a program that directly or indirectly influence the value of a variable at some point in a program. Forward computing slicing algorithms operate in tandem with program execution, they compute the slice of every executing statement and at any point during execution they have available the slices associated with every active object in the program. This makes them well suited to be integrated with our tool whether it is deployed online or used offline for interactive debugging. This dissertation proposes a dynamic slicing algorithm, which could be characterized as forward computing, precise and applicable to unstructured programs, which is an original contribution in itself. Note that the use of this algorithm is not limited to detecting illegal flows; it can also be used to diagnose normal defects.

Figure 4 shows Java code with a defect at line L1 that causes unconditional flow of secure data through line L4. The statements comprising the dynamic slice at line L4, with secureData set to either false or true, are shown underlined. Clearly, this dynamic slice enhances the chances of the programmer to identify the root cause of the illegal flow, i.e., the defect at line L1, since L2 will be part of the slice even if secureData is false. Notice that the static slice at line L4 is less effective in revealing the defect at line L1 since it additionally includes line L3.

1.3 Information flow and slice profiling

Observation-based testing is an approach in which executions of a program are profiled and then analyzed using cluster analysis and sampling methods to identify
Figure 4 – Java code with a defect that causes an unconditional information flow of secure data

unusual or suspicious executions for manual auditing. This technique can be used to detect ordinary failures as well as security attacks. It can also be the basis for test-case filtering. This dissertation contributes to observation-based testing research by devising
two profiling techniques that capture relatively more complex interactions between the components of an executing program namely, information flow profiling and slice profiling. Figure 5 shows the high-level architecture of the observation-based testing process. The original Java byte code program is instrumented to generate execution profiles that are based on various schemes including basic blocks, function calls, control flows and data flows. The instrumented program is replayed using captured executions from the field (see Section 2.4). The execution profiles are analyzed using cluster analysis, and the unusual executions are manually audited. Dickinson et al have empirically shown that those unusual executions are likely to correspond to defects [15][16] and hypothesize that they are also likely to correspond to intrusions.

Leon and Podgurski [47] have empirically shown that profile type has little or no effect on the efficiency of test-case filtering using OBT but considerable effect on the efficiency of test-case filtering using coverage-based techniques. Their results showed that when using coverage-based techniques, finer granularity profiles detected more defects, e.g. the tests that cover all occurring control flows detected more defects than the tests that covered all executing basic blocks. This dissertation extends their experiments by using more subject programs and new profiling techniques, i.e., data flow, information flow and slice profiles.

Finally, we expect that information flow profiles would enhance the capabilities of observation-based testing for anomaly detection especially when the intrusive behavior involves illegal information leaks. This dissertation presents preliminary experimental work related to that issue.
Figure 5 - High-level architecture of the observation-based testing process.
1.4 Outline

The remainder of this dissertation is organized as follows. The next chapter surveys related work on information flow analysis, dynamic slicing, observation-based testing and capture/replay. Chapter 3 presents the definitions and equations needed to describe our algorithms. Chapters 4 and 5 present our algorithms. Chapter 6 discusses our implementation for Java byte code programs. Chapter 7 presents a number of case studies and discusses performance. Chapter 8 presents our approach to implicit flow detection. Chapter 9 evaluates the use of information flow and slice profiling in test-case filtering. Chapter 10 discusses the correlation between the length and strength of an information flow. Chapter 11 views our work from an intrusion detection perspective. Chapter 12 discusses future work and Chapter 13 is the Conclusion.
Chapter 2

Related work

This dissertation incorporates and integrates several research areas, namely: information flow analysis, dynamic program slicing, observation-based testing and capture/replay. Note that capture/replay which is an enabling technology for observation-based testing will only be discussed in this section. Any work related to this area will be deferred to the future.

2.1 Information flow analysis

Lampson motivated the work on information flow analysis by describing the problem and listing a number of possible information leaks [46]. Fenton proposed an abstract machine called the Data Mark Machine in which each variable, including the program counter (PC), is tagged with a security level [21]. A write to a variable is inhibited if its level is lower than the PC level. (Note that the instruction set of the machine contains an instruction to reset the PC level.) Jones and Lipton proposed a similar mechanism in which only the output of a program is checked for illegal flows [39] and they proved that it is not even possible to construct a static mechanism that rejects exactly the insecure executions of a program. Brown and Knight presented a hardware mechanism that tracks security levels on a per-word basis [10]. The Perl scripting language provides a special execution mode called taint mode, in which all user supplied input is treated as suspicious unless the programmer explicitly approves it. Taint mode is very effective at preventing flows from user supplied input data to
potentially dangerous system calls, but is unable to detect or prevent any other type of flows, including ones resulting from control dependence.

Compile-time mechanisms are usually more secure but less precise. Denning and Denning [12][13][14] proposed a static certification mechanism for verifying a program’s compliance with an information flow policy. Language-based type checking systems were proposed in [54][55][67], where in such systems every program expression is assigned a security type in addition to its ordinary type. In type checking a program, the compiler ensures that the program cannot contain illegal information flows at run-time.

2.2 Dynamic program slicing

Tip [75] provides a survey on both static and dynamic program slicing techniques. Dynamic program slicing was first introduced by Korel and Laski [44]; their approach computes executable slices for structured programs. Korel later proposed another algorithm that computes executable slices for possibly unstructured programs by omitting removable blocks [42]. Agrawal and Horgan used Dynamic Dependence Graphs (DDG) and a reduced version of them to compute dynamic slices [1]. Zhang et al proposed three new, more precise algorithms for dynamic slicing, the most efficient of which is called the Limited Processing algorithm and involves preprocessing the execution trace to support demand-driven analysis [86]. All of the aforementioned algorithms are based on “backward” analysis, which requires the execution trace to be available a priori. Forward computing dynamic slicing and information flow analysis algorithms, which are the focus of this work, do not require the execution trace to be available a priori; hence they are more suitable for online analysis, interactive debugging, and long running
programs where recording the execution trace is not feasible. Korel and Yalamanchili were the first to propose such an approach, which is applicable to structured programs [45]. In this approach, a slice is constructed from the original program by omitting removable blocks (of a simpler form than in [42]). The decision to remove a block from a dynamic slice is made by the algorithm when the execution trace traverses the exit point of that block. In [9][20], another forward computing slicing algorithm was proposed for possibly unstructured C programs. However, this algorithm employs very imprecise rules for capturing control dependences involving goto, break, and continue statements. For example, when a goto statement is executed, the slices of the target statement and of all statements that execute following it contain the goto and the statements it is dependent on. The algorithm’s precision is further reduced by the fact that it determines the variables used by a given statement statically. Also note that this algorithm assumes that the execution trace is initially saved on disk, i.e., it executes offline. Chapter 7 presents an example that contrasts the slices computed using the algorithm in [20] and the slices computed using our proposed algorithm. Song et al [70] extended the forward computing algorithm in [45] to object-oriented programs. Zhang et al [87] analyzed the characteristics of dynamic slices, such as their reappearance, overlapping and clustering. They devised a forward computing algorithm based on reduced order binary decision diagrams (roBDD), which exploited the identified characteristics and exhibited tremendous space efficiency. However, the time required to build the roBDD of an execution trace inhibits their algorithm of being used online or interactively; similar to backward computing algorithms their implementation initially saves the execution trace.

Akgul et al [6] presented a reverse code generation algorithm that generates a reverse
version of a program by constructing the inverses of the assembly instructions in the program. They showed how their algorithm could be exploited to extract dynamic slices during reverse execution starting at the slicing criterion.

Korel and Laski [43] defined a relation called potential influence which relates a control statement to other statements it might affect had it been evaluated differently. They used this relation in an interactive algorithm that assists in identifying faulty statements. This algorithm requires the user’s feedback at intermediate stages of the fault identification process. Agrawal and Horgan [5] used the potential influence relation to build what they call relevant slices, i.e., dynamic program slices that take into consideration potential influences; their algorithm is based on extending the Dynamic Dependence Graph whose size is unbounded. They proposed that coverage of relevant slices involving certain (limited) code changes be used as a basis for test suite reduction in regression testing. Gyimothy et al [28] presented an earlier version of the algorithm in [9] that also computes relevant slices based on extending Program Dependence Graphs. Their algorithm is forward computing but only handled a single procedure, scalar variables and no unstructured control. Wang et al [79] presented a backward computing algorithm that is also based on augmenting the PDG for computing relevant slices. They discussed the use of relevant slices in the detection of what they call omission errors in Java byte code programs. There is a clear parallel between the potential influence relation and the implicit flow relation we described earlier (surprisingly there is no reference in [43] to the work presented in [13]). Thus, our approach to detecting implicit flows (Chapter 8) is related to the aforementioned work on relevant slices. Our approach uses program transformations and as in [28] it uses a forward computing algorithm, but it
can additionally handle inter-procedural flows, unstructured constructs, and control statements involving object and array references.

2.3 Observation-based testing

Observation-based testing is an approach to reducing the manual effort required to detect “ordinary” software defects [15][16][48][62][63]. It involves applying multivariate data analysis, data mining, or machine learning techniques to profiles of program executions in order to identify a subset of executions to be audited for conformance to requirements. It has been shown that failures in operational software are often associated with unusual inputs or cause unusual program behavior. Therefore, auditing unusual executions is effective in detecting defects.

The efficiency of observation-based testing is closely coupled with the profiling techniques used in generating the execution profiles. Some techniques are good at revealing specific classes of defects more than others. In some cases combining various profiling techniques produces better results than using them individually.

Leon and Podgurski presented an empirical comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases [47]. Their results indicated that distribution-based techniques (OBT) can be as efficient as or more efficient for revealing defects than coverage-based techniques [83][17][18], but that the two kinds of techniques are also complementary in the sense that they find different defects.

2.4 Capture/Replay
Considerable work has been done in the area of capture/replay, both commercially and in the academic and research centers. The most prevalent commercial application has been regression testing of graphical user interfaces [53][64][73], where a sequence of GUI actions is recorded in the form of a test script. The script is replayed to exercise later versions of the GUI then compare actual to expected outputs. Non-GUI inputs are typically ignored, it is assumed that certain inputs such as contents of files remain available and are unchanged between capture and replay.

Other commercial applications lack the replay capability but capture valuable data about an execution run such as symbol data, network and database resource usage [25][31]. Several researchers focused on the problem of deterministic replay [11][65][68], they were successful to some extent but each of their solutions suffered from serious drawbacks. In addition to exhibiting tremendous performance problems, both of RecPlay [65] and Eraser [68] require the modification of the Unix operating system. Dejavu [11] performs relatively better but requires a specialized JVM and is not publicly available.

The jRapture prototype [72] is a capture/replay tool for Java applications. Figure 6 presents a high level architectural overview of jRapture. In its current state jRapture handles capture/replay of FileInputStream, RandomAccessFile, GUI events, system time queries, random number generation and system properties. Its observed performance seems acceptable but it lacks support for the following:

1) Network I/O classes (Socket etc.)
2) Distributed applications
3) Fault-tolerance (in case the target application fails or misbehaves)
4) Reliable GUI replay

5) Deterministic replay of thread schedules

In an effort to exploit its existing capabilities, we ported jRapture to various versions of the JDK (1.2, 1.3, 1.4) and operating systems (RedHat 6.1, 6.2, 7.2 and Windows 2000). The JDK 1.4 version running on Windows 2000 exhibited the most stability. The GUI capture/replay mechanism was replaced to use the Java ‘robot’ class. The assumptions made by the thread and component identification mechanisms in jRapture did not always hold. Alternative mechanisms had to be formulated. Without the aforementioned changes and a number of additional bug fixes, none but the most trivial applications could be handled by jRapture.
Figure 6 – jRapture high-level architecture.
Chapter 3

Definitions

The theoretical basis of our algorithms presented in Chapters 4, 5 and 8 lies in the formal model of program dependences discussed in [60][61]. Here we present a number of definitions and equations that form the basis of our algorithms. First, background definitions are provided. Second, dynamic control dependence and dynamic data dependence are defined. Third, direct and indirect “influence” relations are defined. Finally, the influence relations are used to derive the equations for information flows and dynamic slices.

3.1 Background definitions

Our DIFA and slicing algorithms are primarily based on the graph-theoretic definitions of control and data dependence. Therefore, we start by providing definitions related to graphs.

**Definition 1**: A digraph is a pair \((V(G), E(G))\), where \(V(G)\) is any finite set and \(E(G)\) is a subset of \(V(G)\times V(G)-\{(v, v) \mid v \in V(G)\}\). If \((u, v) \in E(G)\) then \(u\) and \(v\) are adjacent vertices. The in-degree of vertex \(v\) is the number of predecessors of \(v\), and the out-degree of \(v\) is the number of successors of \(v\).

**Definition 2**: A walk \(W\) in a digraph \(G\) is a sequence of vertices \(v_1v_2...v_n\) such that \(n \geq 0\) and \((v_i, v_{i+1}) \in E(G)\) for \(i = 1, 2, ..., n-1\).
**Definition 3:** A control flow graph \( G \) is a directed graph that satisfies each of the following conditions:

1) The maximum out-degree of the vertices of \( G \) is at most two.

2) \( G \) contains two distinguished vertices: the initial vertex \( v_I \), which has indegree zero, and the final vertex \( v_F \), which has out-degree zero.

3) Every vertex of \( G \) occurs on some \( v_I, v_F \) walk.

A walk in a control flow graph beginning with \( v_I \) is called an initial walk.

Dynamic dependences are defined in the context of execution traces, which we define below.

**Definition 4:** An execution trace \( T \) is a sequence of elements called actions. An action is an executing program statement (or basic block). An action at position \( k \) in \( T \) is denoted \( T(k) \) or \( s^k \) where \( s \) is the executing statement. For a given statement \( s \), \( s^\lambda \) denotes the most recent action involving \( s \), i.e., the most recent execution of \( s \). A sub-trace of \( T \), denoted \( T(k, m) \), is the sub-sequence of actions starting at \( T(k) \) and ending at \( T(m) \).

Note that given the correspondence between execution traces and initial walks in the programs’ control flow graphs, we will equate traces with initial walks and actions with vertex occurrences.

### 3.2 Dynamic control dependence

The concept of postdominance plays an important role in characterizing control dependence between program statements; it is defined first.
Definition 5: Let $G$ be a control flow graph. A vertex $u \in V(G)$ postdominates a vertex $v \in V(G)$ iff every $v.v_f$ walk in $G$ contains $u$; $u$ properly postdominates $v$ iff $u \neq v$ and $u$ postdominates $v$.

Definition 6: Let $G$ be a control flow graph. The immediate postdominator of a vertex $v \in \{V(G) - v_f\}$, denoted $ipd(v)$, is the vertex that is the first proper postdominator of $v$ to occur on every $v.v_f$ walk in $G$.

The immediate postdominance relation is used repeatedly in our algorithms presented in Chapters 4 and 8.

Informally, in the static context, a statement $s$ is control dependent on a statement $t$ if the control structure of the program indicates that $t$ decides, via the branches it controls, whether $s$ is executed or not. Ferrante et al [22] provided the classical graph-theoretic definition of static control dependence, which is applicable to both structured and unstructured programs. We call this relation direct control dependence and we formally define it as follows.

Definition 7: Let $u, v$ be vertices of a control flow graph $G$. Then $v$ is directly control dependent on $u$, denoted $v DCD u$, iff $u$ has successors $v'$ and $v''$ such that $v$ postdominates $v'$ but not $v''$. The set of vertices that $v$ is directly control dependent on is denoted $DCD(v)$.

In [14], Denning defined a relation that is the transitive closure\(^3\) of the $DCD$ relation, which we call control dependence.

\(^3\) Let $G$ be a digraph with vertices $V(G)$ and edges $E(G)$. The transitive closure of $G$ is a graph $G'$ with vertices $V(G)$ and edges $E'(G')$ such that for all $v, w$ in $V(G)$ there is an edge $(v, w)$ in $E'(G')$ iff there is a walk from $v$ to $w$ in $G$, i.e. $w$ can be reached from $v$. 

23
**Definition 8**: Let $u, v$ be vertices of a control flow graph $G$. Then $v$ is *control dependent* on $u$, denoted $v \ CD u$, if there is a $u$-$v$ walk in $G$ that does not contain $ipd(u)$; such a walk is said to *demonstrate* that $v \ CD u$. The set of vertices that $v$ is control dependent upon is denoted $CD(v)$.

Furthermore, in [61], Podgurski characterized direct control dependence as follows.

**Definition 9**: Let $G$ be a control flow graph, and let $u, v \in V(G)$. Vertex $v$ is *directly control dependent* on vertex $u$, denoted $v \ DCD u$, iff $\exists$ a walk $uWv$ in $G$ that such that both of the following are true:

1) $uWv$ demonstrates that $v \ CD u$

2) $v$ is not control dependent on any vertex of $W$.

The walk $uWv$ is said to demonstrate that $v \ DCD u$.

Table 1 shows $ipd(v)$, $CD(v)$ and $DCD(v)$ for all the vertices in the control flow graph of Figure 7. Next, we define the new $DDynCD$ relation, which is the dynamic counterpart of the $DCD$ relation. The dynamic control dependence relation, which is the dynamic counterpart of the $CD$ relation, is the transitive closure of the $DDynCD$ relation.

**Definition 10**: Let $s^k$ and $t^m$ be two actions in an execution trace $T$, where $k < m$ and $s^k$ is a predicate action. Then $t^m$ is *directly dynamically control dependent* on $s^k$, denoted $t^m \ DDynCD s^k$, iff the subtrace $T(k, m)$ demonstrates that $t \ DCD s$. The unique predicate action (if any) that $t^m$ is directly dynamically control dependent on is denoted $DDynCD(t^m)$.

Intuitively, $DDynCD(t^m)$ is the *most recent* predicate action $s^k$ to occur prior to action $t^m$ such that $t \ DCD s$. Note that the $DDynCD$ relation is applicable to programs with arbitrary control structure. For example, for the control flow graph of Figure 7 and the
execution trace $T = <v_i, v_1, v_3, v_4, v_1, v_2, v_4, v_5, v_7, v_2, v_4, v_5, v_6, v_8, v_f>$ the $DDynCD$ relationships are shown in the second column of Table 2.

3.3 Dynamic data dependence

Informally, an action $t^m$ in an execution trace $T$ is data dependent on an action $s^k$ if there is a sequence of assignments in $T$ that propagates data from $s^k$ to $t^m$. In order to model dynamic data dependences between actions the following definition is needed.

**Definition 11:** Given an execution trace $T$ and an action $s^k$. $D(s^k)$ denotes the set of variables defined (assigned a value) by $s^k$. $U(s^k)$ denotes the set of variables used (having their values referenced) by that action. Also, given a set of actions $A$ and sub-trace $T(k_u, k_m)$ we have:

```
<table>
<thead>
<tr>
<th>v</th>
<th>ipd(v)</th>
<th>CD(v)</th>
<th>DCD(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi</td>
<td>v_f</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v1</td>
<td>v_4</td>
<td>v_6, v_5, v_6, v_7</td>
<td>v_6</td>
</tr>
<tr>
<td>v2</td>
<td>v_4</td>
<td>v_1, v_4, v_5, v_6, v_7</td>
<td>v_1, v_7</td>
</tr>
<tr>
<td>v3</td>
<td>v_4</td>
<td>v_1, v_4, v_5, v_6, v_7</td>
<td>v_1</td>
</tr>
<tr>
<td>v4</td>
<td>v_5</td>
<td>v_4, v_5, v_6, v_7</td>
<td>v_5</td>
</tr>
<tr>
<td>v5</td>
<td>v_8</td>
<td>v_5, v_6, v_7</td>
<td>v_7</td>
</tr>
<tr>
<td>v6</td>
<td>v_8</td>
<td>v_5, v_6, v_7</td>
<td>v_5</td>
</tr>
<tr>
<td>v7</td>
<td>v_8</td>
<td>v_5, v_6, v_7</td>
<td>v_5, v_6</td>
</tr>
<tr>
<td>v8</td>
<td>v_f</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vf</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Table 1 - $ipd(v)$, $CD(v)$ and $DCD(v)$ relationships of the control flow graph in Figure 7**
Formally, dynamic data dependence is the transitive closure of the \( \text{DDynDD} \) relation defined below.

**Definition 12:** Let \( s^k \) and \( t^m \) be two actions in an execution trace \( T \), where \( k < m \). Then \( t^m \) is directly dynamically data dependent on \( s^k \), denoted \( t^m \ \text{DDynDD} \ s^k \), iff

\[
(D(s^k) \cap U(t^m)) - D(T(k, m - 1)) \neq \emptyset
\]

<table>
<thead>
<tr>
<th>( T )</th>
<th>( \text{DDynCD} )</th>
<th>Algorithm actions (tagged by line number)</th>
<th>( \text{CDSTACK} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>0</td>
<td>7: ( \text{DDynCD}(v_1) = 0 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( v_1 )</td>
<td>0</td>
<td>7: ( \text{DDynCD}(v_1) = 0 \rightarrow 13: \text{pushes} v_1 )</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>( v_1 )</td>
<td>5: ( \text{DDynCD}(v_3) = v_1 )</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>0</td>
<td>2: ( \text{pops} v_1 \rightarrow 7: \text{DDynCD}(v_4) = 0 \rightarrow 13: \text{pushes} v_4 )</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>( v_1 )</td>
<td>( v_4 )</td>
<td>5: ( \text{DDynCD}(v_1) = v_4 \rightarrow 13: \text{pushes} v_1 )</td>
<td>( v_4 \rightarrow v_1 )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>( v_1 )</td>
<td>5: ( \text{DDynCD}(v_2) = v_1 )</td>
<td>( v_4 \rightarrow v_1 )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>( v_4 )</td>
<td>2: ( \text{pops} v_1 \rightarrow 5: \text{DDynCD}(v_4) = v_4 \rightarrow 11: \text{pops} v_4 \rightarrow 13: \text{pushes} v_4 )</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>0</td>
<td>2: ( \text{pops} v_4 \rightarrow 7: \text{DDynCD}(v_5) = 0 \rightarrow 13: \text{pushes} v_5 )</td>
<td>( v_5 )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>( v_5 )</td>
<td>5: ( \text{DDynCD}(v_7) = v_5 \rightarrow 11: \text{pops} v_5 \rightarrow 13: \text{pushes} v_7 )</td>
<td>( v_7 )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>( v_7 )</td>
<td>5: ( \text{DDynCD}(v_2) = v_7 )</td>
<td>( v_7 )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>( v_7 )</td>
<td>5: ( \text{DDynCD}(v_4) = v_7 \rightarrow 13: \text{pushes} v_4 )</td>
<td>( v_7 \rightarrow v_4 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>( v_7 )</td>
<td>2: ( \text{pops} v_4 \rightarrow 5: \text{DDynCD}(v_5) = v_7 \rightarrow 11: \text{pops} v_7 \rightarrow 13: \text{pushes} v_5 )</td>
<td>( v_5 )</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>( v_5 )</td>
<td>5: ( \text{DDynCD}(v_6) = v_5 \rightarrow 11: \text{pops} v_5 \rightarrow 13: \text{pushes} v_6 )</td>
<td>( v_6 )</td>
</tr>
<tr>
<td>( v_8 )</td>
<td>0</td>
<td>2: ( \text{pops} v_6 \rightarrow 7: \text{DDynCD}(v_8) = 0 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( v_F )</td>
<td>0</td>
<td>7: ( \text{DDynCD}(v_F) = 0 )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

**Table 2** - Sample trace in the CFG of Figure 7 with corresponding \( \text{DDynCD} \) relationships. Column 3 shows the steps taken by the algorithm in Figure 8. Column 4 shows the state of \( \text{CDSTACK} \) with top of stack displayed to the right.
The set of actions that \( t^m \) is directly dynamically data dependent on is denoted \( DDynDD(t^m) \).

Informally, \( t^m \) \( DDynDD \) \( s^k \) iff \( t^m \) uses a variable or object that was last defined by \( s^k \). The \( DDynDD \) relation models both intra-procedural and inter-procedural data dependences. The latter occur when an execution trace spans different functions and data defined in one function is used in another.

### 3.4 Direct influence and influence

In addition to the \( DDynCD \) and \( DDynDD \) relations, we identify three other kinds of dynamic dependences between actions, each of which is inter-procedural. The first, \( ParamD \), is needed to model the use of a value passed by a formal parameter. The second, \( ReturnD \), is needed to model the use of a value returned by a return statement. The third, \( InterprocCD \), is needed to model inter-procedural control dependence on a calling method’s invoke statement (this is similar to the static entry-dependence effect described in [69]). Their formal definitions are presented below.

**Definition 13:** Let \( m \) and \( m' \) be methods and let \( T \) be an execution trace with actions \( s^k \) and \( t^o \) such that \( t^o \) occurs during a call of \( m' \) by \( m \). Then \( t^o \) is parameter dependent on \( s^k \), denoted \( t^o \) \( ParamD \) \( s^k \), iff (1) \( t^o \) uses a formal parameter \( p' \) of \( m' \), which corresponds to actual parameter \( p \) in \( m \), before \( p' \) is defined in \( m' \) and (2) \( s^k \) is the last definition of \( p \) prior to \( t^o \). The set of actions that \( t^o \) is parameter dependent on is denoted \( ParamD(t^o) \).

**Definition 14:** Let \( m \) and \( m' \) be methods and let \( T \) be an execution trace with actions \( s^k \) and \( t^o \) such that \( s^k \) occurs during a call of \( m' \) by \( m \) and \( t^o \) occurs in \( m \) after this call completes. Then \( t^o \) is return dependent on \( s^k \), denoted \( t^o \) \( ReturnD \) \( s^k \), iff \( s \) is a return
statement and \( t^n \) uses the value returned to \( m \) by \( s^k \). The set of actions that \( t^n \) is return dependent on is denoted \( \text{ReturnD}(t^n) \).

**Definition 15:** Let \( m \) and \( m' \) be methods and let \( T \) be an execution trace with actions \( s^k \) and \( t^n \) such that \( t^n \) occurs during a call of \( m' \) by \( m \). Then \( t^n \) is *inter-procedurally control dependent* on \( s^k \), denoted \( t^n \text{ InterprocCD } s^k \), iff \( s^k \) is the action in \( m \) that invoked the call to \( m' \).

The combination of the aforementioned five types of dependences comprises what we call “direct influence” which we formally define.

**Definition 16:** Let \( T \) be an execution trace and let \( s^k \) and \( t^m \) be actions in \( T \), with \( k < m \). Then \( s^k \) *directly influences* \( t^m \), denoted \( s^k \text{ DInfluence } t^m \), iff any of the following conditions is true:

1. \( t^m \text{ DDynDD } s^k \)
2. \( t^m \text{ DDynCD } s^k \)
3. \( t^m \text{ ParamD } s^k \)
4. \( t^m \text{ ReturnD } s^k \)
5. \( t^m \text{ InterprocCD } s^k \)

The set of actions that \( t^m \) is directly influenced by is denoted \( \text{DInfluence}(t^m) \) which can take the following form:

\[
\text{DInfluence}(t^m) = \text{DDynDD}(t^m) \cup \text{DDynCD}(t^m) \cup \text{ReturnD}(t^m) \cup \text{ParamD}(t^m) \cup \text{InterprocCD}(t^m)
\]

The \( \text{Influence} \) relation represents the different ways, both direct and indirect, that one statement can influence the execution of another. Formally, it is simply the transitive closure of the \( \text{DInfluence} \) relation.
Definition 17: Let $T$ be an execution trace and let $s^k$ and $t^m$ be actions in $T$ with $k < m$. Then $s^k$ influences $t^m$, denoted $s^k$ Influence $t^m$, iff there is a sequence $a_1, a_2, \ldots, a_n$ of actions in $T$ such that $a_1 = s^k$, $a_n = t^m$, and for $i = 1, \ldots, n - 1$, $a_i$ DInfluence $a_{i+1}$. The set of actions that influence $t^m$ is denoted $\text{Influence}(t^m)$. It follows that

$$\text{Influence}(t^m) = \text{DInfluence}(t^m) \cup \bigcup_{s^k \in \text{DInfluence}(t^m)} \text{Influence}(s^k)$$

3.5 Dynamic information flow analysis and dynamic slicing

Informally, information flow occurs from object $y$ (source) to object $x$ (target or sink) whenever information stored in $y$ is propagated directly or indirectly to object $x$. Also, let $e$ be the execution of a sequence of statements, if by observing the value of $x$ after $e$ one can deduce information about $y$ before $e$ then information flow occurred from $y$ to $x$ during $e$.

Dynamic information flow analysis is closely related to dynamic slicing in that they both examine the execution trace to identify dynamic control and data dependence relationships. Whereas dynamic information flow analysis is concerned with identifying the set of all variables or objects that influence a variable or object at an action $s^k$, dynamic slicing is concerned with identifying the set of all statements that influence $s^k$ or that influence a variable or object at $s^k$.

Definition 18: Let $T$ be an execution trace, let $s^k$ and $t^m$ be actions in $T$ with $k < m$, let $x$ and $y$ be variables such that $x$ is used at $s^k$ and $y$ is defined at $t^m$, respectively. Then information flows from $x$ at $s^k$ to $y$ at $t^m$ iff $s^k$ Influence $t^m$. The set of variables or objects from which information flows to $y$ at $t^m$ is the set
Definition 19: Let $T$ be an execution trace and let $t^m$ be an action in $T$. The dynamic slice for $t^m$ is the set of statements

$$DynSlice(t^m) = \{t\} \cup \{s : \exists k [s^k \in DInfluence(t^m)]\}$$

Our dynamic slicing algorithm is based on the following inductive characterization of $DynSlice(t^m)$:

$$DynSlice(t^m) = \{t\} \cup \bigcup_{s^k \in DInfluence(t^m)} DynSlice(s^k) \quad (II)$$

It is often useful to consider the dynamic slice for a variable just before or after an action is executed [81]. The dynamic slice of a variable or object $x$ just before execution of action $t^m$ is simply the slice of the last action before $t^m$ that defined $x$. 

$$InfoFlow(t^m) = U(t^m) \cup U(\text{Influence}(t^m))$$

where $U(t^m)$ is the set of variables or objects used at $t^m$ and $U(\text{Influence}(t^m))$ is the set of variables or objects used at the actions that influence $t^m$.

Our DIFA algorithm is based on the following inductive characterization of the set $InfoFlow(t^m)$:

$$InfoFlow(t^m) = U(t^m) \cup \bigcup_{s^k \in DInfluence(t^m)} InfoFlow(s^k) \quad (I)$$

Note that to compute the variables or objects from which information flows to a variable or object $y$ at an action $t^m$ that uses but does not define $y$, it suffices to compute $InfoFlow(d^n)$ as above for the last definition $d^n$ of $y$ before $t^m$.

The dynamic slice for an action $t^m$ is the set of statements executed in actions that influence $t^m$, plus $t$ itself.
Chapter 4

Direct dynamic control dependence algorithm

In this chapter we present an algorithm for computing the direct dynamic control dependence relation $DDynCD$. It is invoked as part of our information flow analysis and dynamic slicing algorithms, which are presented in Chapter 5.

4.1 $DDynCD$ algorithm

The algorithm is shown in Figure 8, it is applicable to both structured and unstructured programs. The algorithm applies to an individual method $m$ and requires that its control flow graph $G(m)$ and immediate postdominators have been computed beforehand, which can be done statically or when a class is loaded. Another important data structure used by the algorithm is a stack, denoted $CDSTACK(m)$, which stores references to predicate actions whose “dynamic control scope” has not yet been exited. The following lists the basic steps of the algorithm:

Step 1. When a predicate action $s^\downarrow$ is reached, push it onto $CDSTACK(m)$; its dynamic control scope has just been entered.

Step 2. When the $ipd$ of a predicate action $s^\downarrow$ is reached, pop it off $CDSTACK(m)$; its dynamic control scope has just been exited.

Step 3. Current action is directly control depend on the predicate action sitting on top of $CDSTACK(m)$ (denoted $TOS$ from now on).
function \texttt{DDynCD}(s^{\lambda}, \texttt{CDSTACK}(m))

/*
ipd(s): immediate post dominator of a statement s
TOS(m): top of \texttt{CDSTACK}(m)
*/

1 \textbf{if} \neg \text{Empty}\texttt{(CDSTACK}(m)\texttt{)} \text{ and } s = \text{ipd}\texttt{(TOS}(m)\texttt{)} \text{ then}
2 \quad \text{pop CDSTACK}(m)
3 \textbf{endif}

4 \textbf{if} \neg \text{Empty}\texttt{(CDSTACK}(m)\texttt{)} \text{ then}
5 \quad \texttt{DDynCD}(s^{\lambda}) = \texttt{TOS}(m)
6 \textbf{else}
7 \quad \texttt{DDynCD}(s^{\lambda}) = \text{null}
8 \textbf{endif}

9 \textbf{if} s \text{ is a decision statement} \text{ then}

10 \textbf{if} \neg \text{Empty}\texttt{(CDSTACK}(m)\texttt{)} \text{ and } \text{ipd}(s) = \text{ipd}\texttt{(TOS}(m)\texttt{)} \text{ then}
11 \quad \text{pop CDSTACK}(m)
12 \textbf{endif}

13 \quad \text{push } s \text{ onto } \texttt{CDSTACK}(m)
14 \textbf{endif}

\textbf{return } \texttt{DDynCD}(s^{\lambda}) ;

\textbf{Figure 8 - Direct Dynamic Control Dependence Algorithm}

Step 4. When a predicate action \(s^{\lambda}\) is reached such that \text{ipd}(s) is the same as the \text{ipd} of \text{TOS}, pop \text{TOS}. This step ensures that the size of \text{CDSTACK}(m) is bounded by the number of decision nodes in \text{G}(m) as discussed next.

As a result of loops, an arbitrary number of dynamic control scopes may be “open” at a given point during execution of \text{m}. Step 4 above, ensures that the size of \text{CDSTACK}(m)
is bounded by the number of decision vertices in $G(m)$, only the latest action in a sequence of predicate actions involving predicates with the same immediate postdominator is kept on $CDSTACK(m)$. This is permissible because: (1) any trace can be decomposed into a sequence of $m \geq 1$ segments, the last $m - 1$ of which demonstrate a chain of $DDynCD$ relationships between statements and (2) the chain of $DDynCD$ relationships can be decomposed into blocks of vertices with the same immediate postdominator. These results are stated formally and proved by Podgurski in [51].

The inputs to the algorithm are the executing action $s^\hat{\lambda}$ and $CDSTACK(m)$ where $m$ is the executing method. The output is $DDynCD(s^\hat{\lambda})$ or ‘null’.

On line 13 of Figure 8, the algorithm pushes the most recently executed predicate action onto $CDSTACK(m)$ (Step 1). Lines 4-8 set $DDynCD(s^\hat{\lambda})$ to $TOS(m)$ or ‘null’ in case $CDSTACK(m)$ is empty (Step 3). A predicate action is popped off $CDSTACK(m)$ when any of the following occur:

a) Its immediate post dominator is executed, lines 1-3 (step 2).

b) A predicate action having the same immediate post dominator is executed, lines 10-12 (Step 4).

Table 2 shows how computing the $DDynCD$ relationships using the algorithm, leads to the same results as applying the definition of the $DDynCD$ relation. The actions taken at lines 5 and 7 set the value of $DDynCD(s^\hat{\lambda})$, their outcome agree with the entries in column 2. Next, we graphically demonstrate the steps taken by the algorithm and show how omitting Step 4 might result in $CDSTACK(m)$ growing indefinitely.

Given a control flow graph involving two predicate vertices and an arbitrary execution trace = $<1^1, 2^2, 3^3, 5^4, 6^5, 2^6, 3^7, 5^8, 6^9, 2^{10}, 3^{11}, 5^{12}, 6^{13}>$. The following
sequence of figures contrasts the states of $CDSTACK(m)$ when Step 4 is performed and when Step 4 is omitted. Notice how in the latter case $CDSTACK(m)$ tends to grow as vertices 3 and 6 execute. (The currently executing vertex is depicted in dark.)
CDSTACK(m)

CDSTACK(m)
(if omitting Step 4)

vertex 3*

CDSTACK(m)
(if omitting Step 4)

vertex 3*

ipd of verices 3 and 6

ipd of verices 3 and 6
CDSTACK(m)

vertex 3*

CDSTACK(m)
(if omitting Step 4)

vertex 3*

CDSTACK(m)

vertex 6*

CDSTACK(m)
(if omitting Step 4)

vertex 6
vertex 3

ipd of vertices 3 and 6
The diagram illustrates a sequence of operations on a graph, labeled as "CDSTACK(m)". The process involves iterating through vertices and placing them in a stack (CDSTACK(m)) in a specific order. The diagram shows the flow from vertex 1 to vertex 7, with intermediate steps highlighting the process of "ipd of verices 3 and 6" and "if omitting Step 4". The vertices are sequentially placed in the stack, with the final configuration showing vertex 6 and vertex 3.

The steps include:
1. Start at vertex 1.
2. Move to vertex 2.
3. Move to vertex 3.
5. Move to vertex 5.
7. Move to vertex 7.

The diagram also includes a note indicating the vertices 6 and 3 being added to the stack in a specific order. The final stack configuration shows vertex 6, vertex 3, and vertex 3 again, indicating a repeated action or step.
$CDSTACK(m)$

(vertex 3*)

(vertex 3)
(vertex 6)
(vertex 3)

$CDSTACK(m)$ (if omitting Step 4)

$CDSTACK(m)$

(vertex 6*)

$CDSTACK(m)$ (if omitting Step 4)

(ipd of vertices 3 and 6)

(ipd of vertices 3 and 6)
ipd of vertices 3 and 6

CDSTACK(m)

vertex 6*

CDSTACK(m)
(if omitting Step 4)

vertex 6
vertex 3
vertex 6
vertex 3

ipd of vertices 3 and 6

CDSTACK(m)

vertex 3*

CDSTACK(m)
(if omitting Step 4)

vertex 3
vertex 6
vertex 3
vertex 6
vertex 3
1

2

3

4

5

6

7

CDSTACK(m)

vertex 3*

CDSTACK(m)
(if omitting Step 4)

vertex 3

vertex 6

vertex 3

vertex 6

vertex 3

CDSTACK(m)

vertex 6*

CDSTACK(m)
(if omitting Step 4)

vertex 6

vertex 3

vertex 6

vertex 3

vertex 6

vertex 3

ipd of vertices 3 and 6

ipd of vertices 3 and 6
4.2 Time and space complexity

At a given action, the DDynCD algorithm executes no more than a handful of selection and sequential statements. Therefore, the time complexity of DDynCD is $O(1)$ per action.

Each invoked method $m$ in an application will have an instance of $CDSTACK(m)$ per thread (i.e., per execution trace). As discussed earlier, the size of $CDSTACK(m)$ is bounded by the number of decision nodes in $G(m)$. Let $t$ be the number of executing threads and $d$ the number of decision nodes in an application, the worst case space requirements for the DDynCD algorithm is $O(t \cdot d)$. The worst case occurs when the dynamic control scope of all decision nodes in all threads are active at the same time for a given action.

Note that the slicing algorithm described in [87] also computes the DDynCD relation (there termed dynamic control dependence). For each statement $t$, the set $CD(t)$ of predicate statements on which $t$ is statically control dependent is computed. Action $t^m$ is determined to be dynamically control dependent on action $s^k$ if $s \in CD(t)$ and $k$ is the latest time stamp of any executed predicate from $CD(t)$. This algorithm requires $O(n)$ worst-case time per action processed, where $n$ is the number of predicate statements in a method. Our DDynCD algorithm, on the other hand, requires only constant time per action processed, assuming that immediate postdominators are precomputed. Also, the dynamic control dependence chain captured by $CDSTACK$ is potentially useful during debugging.
Chapter 5

Dynamic information flow analysis and dynamic slicing algorithms

In this chapter, we describe our forward-computing algorithms for dynamic information flow analysis and dynamic slicing, which achieve their precision by basing their intra-procedural control dependence sub-algorithm on the $DD_{ynCD}$ relation.

5.1 InfoFlow and Dynslice algorithms

Figure 9 shows a description of the algorithms (integrated together). The algorithms apply equations (I) and (II) sequentially at every action $t^i$ of an executing program and then store the results for subsequent use. In this manner, when applying the

\begin{verbatim}
procedure InfoFlowAndDynSlice (t™)

\end{verbatim}
equations at \( t^n \), all the values they depend on have already been computed and are available. This demonstrates the forward computational characteristic of the algorithms and parallels the algorithm presented in [9][20]. Our slicing algorithm computes more accurate slices because of the precision of its intra-procedural control dependence sub-algorithm (DDynCD) and the fact that it computes data dependences (DDynDD) dynamically as object and array references are resolved.

5.2 Time and space complexity

We now analyze the time and space complexity of our algorithms. Let \( m \) be the number of statements and \( n \) be the number of active objects in a program. In our algorithms, set union operations have the biggest impact on performance. Assume that primitive set operations (e.g. add, contains) used to implement unions have unit cost.\(^4\) Then the worst case time requirement to compute InfoFlow at an action \( t^k \) (excluding the time required to compute it at prior actions) is \( O(n^2) \); the worst case, involving the union of \( n \) sets each with \( n \) objects, occurs when information flows from every object into every other object and \( t^k \) uses all active objects. The worst case time requirement to compute DynSlice at \( t^k \) (excluding the time required to compute it at prior actions) is \( O(n \cdot m) \); the worst case, involving the union of \( n \) sets each with \( m \) statements, occurs when the slice at every object comprise all statements and \( t^k \) uses all active objects.

At any given time, the algorithm stores InfoFlow and DynSlice sets for each active object. Therefore, the worst case storage requirement of InfoFlow is \( O(n^2) \); the worst case occurs when information flows from every object into every other object in the

\(^4\) In our implementation, sets are implemented with hash tables, so such operations have \( O(1) \) cost on average.
program. The worst case storage requirement of DynSlice is \( O(n \cdot m) \); the worst case occurs when the slices of all \( n \) objects comprise all statements.

Note that typically, the tool implementing our algorithms would be configured to track only a handful of objects (see Chapter 7), and its slicing feature would be enabled during debugging only.
Chapter 6

Implementation

We have implemented our information flow analysis and dynamic slicing algorithms in a prototype tool that can be used to test and/or debug an unstructured Java byte code program against an information flow policy. This tool could also be used to detect illegal flows in a deployed system as long as it is not time-critical. This capability and interactive debugging are feasible only with forward computing algorithms. Note that information leaks could be detected without the program slicing capability, but slices are useful for debugging such leaks. In the current version of the tool, the user can define desired information flow policies by programmatically describing the following three entities:

1) The source objects to monitor, referred to as Sources.

2) The sensitive objects among Sources that might be involved in an illegal flow, referred to as SensitiveSources.

3) The sink methods or objects, at which illegal flows are checked, referred to as Sinks.

In illegal flow detection, Sources and SensitiveSources are typically the same. In debugging, SensitiveSources are typically a subset of Sources. The Sinks are typically methods that expose internal application data to the outside world, such as print and send. Note that part of our future work is to provide a user interface for easier configuration of information flow policies.
In conjunction with a conventional debugger⁵, the tool is capable of providing, at any point during execution, information that includes the information flows and slices for global symbols and local symbols in each stack frame. Typically, this information would be examined right after an illegal flow is detected. Currently, when any insecure flows are detected, the tool logs them and the related lengths (see Chapter 10) and slices and then stops program execution. Note that the tool could be configured to generate DIFA and/or slice profiles to be used in offline analyses (see Chapters 9 and 11).

The tool comprises two main components: the Instrumenter and the Profiler. The preliminary step in applying our tool is to instrument the target byte code classes and/or jar files using the Instrumenter, which is based on the Byte Code Engineering Library, BCEL [8]. When applying our tool to a program, one can instrument any subset of program classes (no source code required) and Java libraries. Generally, the more classes that are instrumented, the more accurate but costly the analysis is. The Instrumenter inserts a number of method invocations to the Profiler at some given points of interest. At runtime, the instrumented application invokes the Profiler passing it the necessary information that enables it to monitor information flows and build program slices. Appendix A presents a detailed description of those methods and the data structures they use.

⁵ Note that the integration of our tool with the Eclipse Java IDE (www.ecplise.org) is underway
Chapter 7

Experimental Results

We first provide a detailed discussion of applying our tool to a simple unstructured Java byte code program. Then we present five case studies where the utility of our tool in illegal flow detection is shown. Finally, we assess the performance impact of our tool on a number of Java programs.

7.1 Unstructured Java Byte Code program

We applied our tool to the Java byte code program shown in Figure 10 without defining any information flow policy (i.e., $Sources = \{\text{all objects}\}$, $SensitiveSources = Sinks = \emptyset$). Since Java does not have goto statements, for testing purposes we used the Byte Code Engineering Library (BCEL) [8] to introduce them in the program. Table 3 shows the $InfoFlow$ and $DynSlice$ sets computed by our tool for this program, in addition to $DynSlice^*$, which are the slices computed by the algorithm presented in [9][20]. Notice in Table 3, how the slices in $DynSlice^*$ are never smaller than the slices in $DynSlice$. This is mainly due to the fact that with the algorithm of [9][20], the slices for all actions after a goto action include the goto and all statements in the slice for the goto action. In addition, when computing $DynSlice^*$ for action 18 we assumed that the statement $z = (b) \ ? x : y$; uses variables $b$, $x$ and $y$ and not just $b$ and $x$, since the algorithm in [9][20] statically determines use variables. Note that we did not implement the algorithm in [9][20]; instead we relied on our understanding of it to identify the slices in $DynSlice^*$. 

<table>
<thead>
<tr>
<th>method:</th>
<th>static int method(int mx, int my)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:0: iconstant_1</td>
<td>{</td>
</tr>
<tr>
<td>5:1: istore_0</td>
<td>5: mx = 1;</td>
</tr>
<tr>
<td>6:2: iload_0</td>
<td>6: return mx * my;</td>
</tr>
<tr>
<td>6:3: iconstant_1</td>
<td>}</td>
</tr>
<tr>
<td>6:4: imultiply</td>
<td></td>
</tr>
<tr>
<td>6:5: ireturn</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main:</th>
<th>public static void main(String[] args)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:0: iconstant_1</td>
<td>{</td>
</tr>
<tr>
<td>10:1: istore_1</td>
<td>10: boolean b = true;</td>
</tr>
<tr>
<td>11:2: iconstant_0</td>
<td>11: int x = 0, y = 0, z = 0;</td>
</tr>
<tr>
<td>11:3: istore_2</td>
<td>12: int z1 = 1, z2 = 2, z3 = 3;</td>
</tr>
<tr>
<td>11:4: iconstant_0</td>
<td>14: L1: x++;</td>
</tr>
<tr>
<td>11:5: istore_3</td>
<td>15: L2: y++;</td>
</tr>
<tr>
<td>11:6: constant_0</td>
<td>16: if (z1 &lt; z2)</td>
</tr>
<tr>
<td>11:7: istore</td>
<td></td>
</tr>
<tr>
<td>12:9: constant_1</td>
<td>18: z1 = z2;</td>
</tr>
<tr>
<td>12:10: istore</td>
<td>19: goto L2; // added using BCEL</td>
</tr>
<tr>
<td>12:12: iconstant_2</td>
<td>}</td>
</tr>
<tr>
<td>12:13: istore</td>
<td>21: if (z3 &gt; 0)</td>
</tr>
<tr>
<td>12:15: iconstant_3</td>
<td></td>
</tr>
<tr>
<td>12:16: istore</td>
<td>23: z3 = 0;</td>
</tr>
<tr>
<td>14:18: increment</td>
<td>24: goto L1; // added using BCEL</td>
</tr>
<tr>
<td>15:21: increment</td>
<td>}</td>
</tr>
<tr>
<td>16:24: iload</td>
<td>26: y = method(x, y);</td>
</tr>
<tr>
<td>16:26: iload</td>
<td>27: z = (b)? x : y;</td>
</tr>
<tr>
<td>16:28: if_lessthan</td>
<td>}</td>
</tr>
<tr>
<td>18:31: iload</td>
<td></td>
</tr>
<tr>
<td>18:33: istore</td>
<td></td>
</tr>
<tr>
<td>19:35: goto 14:18</td>
<td></td>
</tr>
<tr>
<td>21:38: iload</td>
<td></td>
</tr>
<tr>
<td>21:40: iflessthan</td>
<td></td>
</tr>
<tr>
<td>23:38: iconstant_0</td>
<td></td>
</tr>
<tr>
<td>23:44: istore</td>
<td></td>
</tr>
<tr>
<td>24:46: goto 15:21</td>
<td></td>
</tr>
<tr>
<td>26:49: iload_2</td>
<td></td>
</tr>
<tr>
<td>26:51: iload_3</td>
<td></td>
</tr>
<tr>
<td>26:53: invokevirtual</td>
<td></td>
</tr>
<tr>
<td>26:56: istore_3</td>
<td></td>
</tr>
<tr>
<td>27:58: iload_1</td>
<td></td>
</tr>
<tr>
<td>27:60: ifequal</td>
<td></td>
</tr>
<tr>
<td>27:63: iload_2</td>
<td></td>
</tr>
<tr>
<td>27:65: goto 27:70</td>
<td></td>
</tr>
<tr>
<td>27:68: iload_3</td>
<td></td>
</tr>
<tr>
<td>27:70: istore</td>
<td></td>
</tr>
<tr>
<td>28:72: return</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10 - Java byte code and its java-like equivalent**

(Byte code statements are identified by the Java line number and offset, e.g. in ‘6:3: iload_1’, 6 is the Java line number, 3 is the byte code offset.)
Table 3 - *DynSlice* and *InfoFlow* respectively show the slices and information flows for the program in Figure 10. Actions corresponding to lines 10, 11, and 12 are omitted. *DynSlice* is computed by [9][20]
7.2 Case Study 1: JConsole

JConsole [37] is a Java Console client written 100% in Java. It is written mainly for use by system administrators and programmers; it is also meant to be plugged into various other applications. JConsole has a basic set of commands such as cd, pwd, ls, history, alias, etc. More commands can be coded and added to it. We added the type command to it, which displays the contents of a text file.

We applied our tool to JConsole in order to detect any leaks from a given set of security-sensitive directories to the user (i.e., to the screen). We defined an information flow policy such that Sources = \{all objects\}, SensitiveSources = \{objects of type java.io.File or java.io.FileInputStream\} associated with files belonging to the sensitive directories\} and Sinks = \{print, println and write methods of the java.io.PrintStream class\}. We issued several ls and type commands in various directories. When the files we were trying to list (using ls) or display (using type) were identified as sensitive (i.e., belonged to a sensitive directory), the tool logged the insecure flows and slices then stopped the execution. For example, when a type command was applied to a sensitive file, the logged InfoFlow set comprised 49 objects, including a reference to the java.io.File object corresponding to the sensitive file. Note that if we defined Sources to be the same as SensitiveSources above, InfoFlow would contain only the sensitive object. The tool also logged DynSlice, which spanned 11 different methods and contained 110 byte code statements. Note that if we instrumented the Java libraries (e.g. java.io) the logged information flows and slices would be much larger.

\footnote{The FileInputStream class in the Java API does not provide access to the name of the file it is associated with. Since some information flow policies require such information, our tool loads a modified version of FileInputStream that does not have this limitation.}
7.3 Case Study 2: MeetingScheduler

We applied our tool to a Java program implementing a meeting scheduler for Pentagon employees (an application we built based on fictitious requirements). Suppose that at the Pentagon there is a class of employees whose activities must be concealed from the outside world and that throughout the Pentagon’s information systems this type of employee is represented by class HSEmployee (high security), whereas LSEmployee (low security) represents other employees. Both HSEmployee and LSEmployee derive from the abstract class Employee, which contains information such as employeeld, name, rank, and expense accountNumber. Suppose that one of the Pentagon’s systems is a meeting scheduler. In order to set up a meeting, an office administrator enters the start time, the expected end time, and the identification numbers of the attendees. When the meeting is expected to last past 8 PM, the system automatically orders food from an outside caterer via a fax server. The fax includes the number of attendees and the expense account number of the employee paying the bill, who is always the highest-ranking attendee. If the recipient of the fax is aware of a pattern exhibited by the expense account numbers of employees of type HSEmployee, in some cases he or she will be able to infer that a high ranking, high security employee is attending a meeting late that night and that perhaps some special operations are being planned.

Figure 11 shows the Java code for an insecure implementation of the described application (only relevant code is shown), which exhibits a potential information leak from an HSEmployee to an object representing a fax machine. Such an insecure implementation might result if: a) the developer was not aware of the security
requirement, b) the security requirement did not exist at the time when the application was developed, or c) the developer failed to check the type of the **PayingEmployee** and the bug remained despite subsequent visual code inspection.

In order to test this implementation using our tool, we instrumented it and defined an information flow policy such that Sources = \{all objects\}, SensitiveSources = \{attributes identifying a high security employee, i.e., name, employeeId or accountNumber of an HSEmployee\} and Sinks = \{methods of the FaxComponent API\}. When executing test cases in which the highest-ranking employee was represented by an HSEmployee object and the meeting ended after 8 PM, the tool detected insecure information flows from an accountNumber attribute to the faxFoodOrder() method. The logged InfoFlow set comprised the following objects: the **PayingEmployee**, the **PayingAccountNumber**, maxRank, v, vAttendeeList, endTime, one instance of accountNumber belonging to an HSEmployee object, several instances of rank belonging to LSEmployee and HSEmployee objects, and two local variables declared in the Employee constructor, which is not shown in Figure 11. The accountNumber instance was determined to be sensitive since it was an attribute identifying an HSEmployee object. That object corresponded to the highest ranking attendee whose expense account number would have been printed on the fax order. Note that if we defined Sources to be the same as SensitiveSources above, InfoFlow would have comprised a single object, i.e., the sensitive accountNumber. Although not necessary for the detection of the illegal leaks, the additional objects in InfoFlow can in some cases be valuable to the analysis.
public class MeetingScheduler
{
    public static void reserveRoom(int size) { ... } // Input: vList is a vector of Employee objects
    // Note: Employee with highest rank will pay for the order
    static void orderFood(Vector vList)
    {
        Employee thePayingEmployee = null;
        String thePayingAccountNumber = null;
        int maxRank = -1;
        // Find the employee with the highest rank
        for (int v = 0; v < vList.size(); v++)
        {
            if (((Employee)vList.elementAt(v)).getRank() > maxRank)
            {
                thePayingEmployee = (Employee)vList.elementAt(v);
                maxRank = thePayingEmployee.getRank();
            }
        }
        thePayingAccountNumber = thePayingEmployee.getAccountNumber();
        String faxText = "Order size:
                        " + vList.size() + "\n                        " + "Bill to account#: " + thePayingAccountNumber;
        FaxComponent.faxFoodOrder(faxText); // outlet to the outside world
    }
    public static void main(String[] args)
    {
        Vector vAttendeeList = new Vector();
        // Populate list with objects of type Employee
        UIComponent.askForEmployeeList(vAttendeeList);
        // Get meeting time information
        Time startTime = UIComponent.askForTime("start time?");
        Time endTime = UIComponent.askForTime("end time?");
        // Reserve a room large enough for all attendees
        reserveRoom(vAttendeeList.size());
        if (endTime.exceeds("20:00")) // meeting to last past 8pm?
        {
            orderFood(vAttendeeList);
        }
    }
}

Figure 11 – Insecure Java code implementing the meeting scheduler
7.4 Case Study 3: DefaultServlet

We applied our tool to the DefaultServlet component of the open source Java application server Apache Tomcat, which exhibits the file disclosure vulnerability described by McClure, et al [52]. This vulnerability has been found in versions of several other widely used Java application servers, including BEA Weblogic and IBM WebSphere. McLure, et al show how this vulnerability in Weblogic’s FileServlet component, which is a servlet designed to serve publicly available text files to clients, can be exploited to reveal Java Server Pages (JSP) source code of another application, which may contain sensitive data. McLure, et al describe how what is learned from the disclosed JSP source code can be used to execute malicious remote commands. Tomcat’s DefaultServlet, which is also intended to serve publicly available text files, can be exploited similarly to reveal the contents of any file within the servlet root directory, simply by passing it the relative path of the file. Initially, in this case study the capability of our tool to detect illegal flows is not utilized; instead we log information flows for later analysis by a security analyst.

Note that a servlet may send response data to a client through a PrintWriter object or through a ServletOutputStream object. Therefore, the write and print methods of these two objects are information sinks that are natural to monitor for illegal information flows. Accordingly, we instrumented DefaultServlet and defined an information flow policy such that Sources = SensitiveSources = {all objects} and Sinks = {write and print methods of PrintWriter and ServletOutputStream}. We issued requests to DefaultServlet to serve various files including ones located in directories not intended for public access, e.g., directories containing JSP source code. In these tests, the
only information flows into a sink object involved a single write method call (with three parameters) associated with a ServletOutputStream. The log recorded the information flows and slices for the ServletOutputStream and each of the write parameters, computed at the time when they were last defined. For each request, there were flows from 17 different source objects into the sink, and there were 45 byte code instructions that affected it. The logged state of 4 of the 17 source objects included the full pathname of the requested file, which should be sufficient for a security analyst to determine whether an illegal request took place. Note that if we opted to instrument all of Tomcat’s libraries or even the Java API library, much more information would have been logged.

Once it is learned that illegal requests are occurring it makes sense to redefine the information flow policy as follows: Sources = SensitiveSources = {objects of type java.io.File or java.io.FileInputStream associated with files not belonging to public directories} and Sinks = {write and print methods of PrintWriter and ServletOutputStream}. This policy would prevent any leaks such as the ones described above.

7.5 Case study 4: Path disclosure

SecurityFocus (www.securityfocus.com) is a vendor neutral web site that provides information related to security, including a vulnerability database. Over one hundred of the reported vulnerabilities in its database involve the insecure disclosure of a server’s root path. The path disclosure vulnerability, which could be used by a remote attacker to launch further attacks, has been found in versions of several widely used Java application servers, including BEA Weblogic and Sun’s IPlanet.
Figure 12 shows the Java code for a servlet, implemented using *Tomcat 3.2.3* Application Server, which is supposed to append user data to the end of a file residing on the web server. In the case when the requested file to be updated does not exist or is not accessible, an exception is thrown and the servlet discloses the full path of the file, which includes the (sensitive) web root directory. In our implementation, the web root directory and other sensitive information are stored in a hash table in the `Config` class.

We defined an information flow policy such that $Sources = \{\text{all objects}\}$, $SensitiveSources = \{\text{Config.configTable}\}$ and $Sinks = \{\text{write} \text{ and print methods of PrintWriter and ServletOutputStream}\}$. When executing test cases in which the file to be updated does not exist, the tool detected insecure information flows from `Config.configTable` to the `out.println` method inside the `catch` block. The *DynSlice* set comprised 32 byte code statements and the *InfoFlow* set comprised the `Config.configTable` object and 15 other objects originating from the `doGet` method and methods belonging to `java.util.Hashtable` and `java.lang.String`.

Based on the defined policy, the `Config.configTable` object in *InfoFlow* was determined to be sensitive. Note that if we defined $Sources$ to be the same as $SensitiveSources$ above, *InfoFlow* would have comprised a single object, i.e., the sensitive `Config.configTable`. 
class Config
{
    private static Hashtable configTable;
    static
    {
        // populate configTable
    }
    public static String getParameter(String name)
    {
        return (String)configTable.get(name);
    }
}

public class UpdateFile extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        PrintWriter out = response.getWriter();
        String strData = request.getParameter("data");
        String strFileName = Config.getParameter("webroot") + request.getPathInfo();
        try {
            FileOutputStream fos = new FileOutputStream(strFileName, true);
            fos.write(strData.getBytes());
            fos.close();
            out.println("Update succeeded");
        } catch (Exception e) {
            out.println("Encountered an error while updating " + strFileName);
        }
    }
}

Figure 12 - Insecure servlet that might disclose the root path of the webserver.

7.6 Case study 5: EmploymentVerifier

We applied our tool to a Java program implementing an employment verification system for health care employees (another application we built based on fictitious requirements). Suppose that at a given hospital, all information systems needing to retrieve employee information would have to invoke the public methods provided by the EmployeeInformation class. One such method is getSummary, which formats basic
employee information and returns it in a string. In addition, an employee who is HIV positive would have its name suffixed with a ‘*’, a convention that supposedly only select hospital employees are aware of. Obviously, the HIV status of an employee must be disclosed to outsiders only in extreme cases.

Suppose that one of the hospital’s systems is an employment verifier. In order to check on the employment status of a hospital employee, an organization such as a mortgage lender or a credit card company can submit a request to the system via voice messaging and expect the response to be sent back automatically via email. If the organization receiving the response is aware of the hospital’s aforementioned convention related to HIV positive employees, they might take discriminatory measures against certain employees.

Figure 13 shows the Java code for an insecure implementation of the described application (only relevant code is shown), it erroneously uses the getSummary method and therefore exhibits a potential information leak from the HIVStatus flag of an employee to an object representing a mail server.

In order to test this implementation using our tool, we defined an information flow policy such that Sources = SensitiveSources = {HIVStatus attribute of an employee} and Sinks = {methods of the MailComponent API}. When executing test cases in which the employee is HIV positive, the tool detected insecure information flows from the Employee.HIVStatus attribute to the sendMessage method. The DynSlice set at sendMessage comprised 54 byte code statements, and the InfoFlow set comprised the single object: employee.HIVStatus.
class EmployeeInformation {
    private static Employee getDetails(String employeeId) { ... }
    public static boolean getEmploymentStatus(String employeeId) { ... }

    public static String getSummary(String employeeId) {
        Employee employee = getDetails(employeeId);

        String employeeSummary = "Name: " + employee.getFirstName() + " " + employee.getLastName();
        if (true == employee.getHIVStatus()) {
            employeeSummary += "*";
        }
        employeeSummary += "ID: " + employeeId + "\n"
        employeeSummary += "Department: " + employee.getDepartment() + "\n"
        employeeSummary += "Supervisor: " + employee.getSupervisor() + "\n";

        return employeeSummary;
    }
}

public class EmploymentVerifier {
    public static void main(String[] args) {
        String id = ...
        String strEmailAddress = ...

        String strMsg = "The employee described below:\n";
        strMsg += EmployeeInformation.getSummary(id) + "\n";
        if (true == EmployeeInformation.getEmploymentStatus(id)) {
            strMsg += "is a current employee of our organization\n"
        } else {
            strMsg += "is no longer an employee of our organization\n"
        }
        MailComponent.sendMessage(strEmailAddress, strMsg);
    }
}

Figure 13 - Insecure Java code implementing the employment verifier

Based on the defined policy, the employee.HIVStatus variable in InfoFlow was determined to be sensitive. Note that when executing test cases in which the employee is not HIV positive, the InfoFlow was empty.

7.7 Time and storage impact

In order to evaluate the performance impact of our tool, we applied it on a number of Java applications. The intention was not to detect insecure information flows but instead to measure the slowdown and the increased memory usage due to using the tool.
Accordingly, we defined an information flow policy such that $Sources = \{ \text{all objects} \}$, $SensitiveSources = Sinks = \emptyset$. The analyzed applications included the five case studies presented earlier, the Java class file disassembler $javap$, the Java compiler $javac$, and nine other freeware or open source applications we downloaded from [35][36]. In most cases typical inputs were used; for user interface applications, we made sure to follow the same pace when inputting data to the original program and the instrumented programs.

Table 4 contrasts the execution times and peak memory usages of the original programs and two other versions of the programs, one instrumented to compute information flows and the second instrumented to compute both information flows and slices. The column labeled ‘# callbacks’ lists the number of times the tool’s profiler interface was called, this number is indicative of the relative lengths of the execution traces.

As expected, in most cases there was a considerable slowdown and storage impact. It is clear that for $processing intensive$ applications (those having relatively large execution traces per request) such as $JarScan$, $Jtidy$, $Diff$, $javac$ and $javap$ with an average slowdown of $3000\%$, it is not feasible to apply our tool online. On the other hand, for interactive and/or non-$processing-intensive$ applications such as $JConsole$, $MeetingScheduler$, $DefaultServlet$ and $JDictionary$, with an average slowdown of $97\%$, it seems feasible to apply our tool either offline or online. Since web and interactive business applications are typically not processing intensive, we believe that they are good candidates to be deployed within our tool.
Table 4 – Time and storage impact for various programs.

# lines: number of Java lines in the instrumented code (* indicates byte code instructions). # callbacks: number of times the tool’s profiler interface was called. Time1: original execution time. Time2: execution time with InfoFlow being computed. Time3: execution time with InfoFlow and DynSlice being computed. Mem1: original peak memory usage. Mem2: peak memory usage with InfoFlow being computed. Mem3: peak memory usage with InfoFlow and DynSlice being computed.

<table>
<thead>
<tr>
<th>Program</th>
<th># lines</th>
<th># callbacks</th>
<th>Time1 (secs)</th>
<th>Time2 (secs)</th>
<th>Time3 (secs)</th>
<th>Mem1 (KB)</th>
<th>Mem2 (KB)</th>
<th>Mem3 (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JConsole</td>
<td>2,049</td>
<td>10,894</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>6,700</td>
<td>16,528</td>
<td>16,572</td>
</tr>
<tr>
<td>Meeting Scheduler</td>
<td>231</td>
<td>9,579</td>
<td>0.4</td>
<td>1.5</td>
<td>1.6</td>
<td>4,640</td>
<td>9,292</td>
<td>9,352</td>
</tr>
<tr>
<td>DefaultServlet</td>
<td>2,342</td>
<td>2,457</td>
<td>0.2</td>
<td>0.5</td>
<td>0.8</td>
<td>26,644</td>
<td>37,102</td>
<td>37,456</td>
</tr>
<tr>
<td>Employment Verifier</td>
<td>184</td>
<td>469</td>
<td>0.1</td>
<td>0.2</td>
<td>0.2</td>
<td>2,740</td>
<td>5,250</td>
<td>5,620</td>
</tr>
<tr>
<td>Path Disclosure</td>
<td>1,296</td>
<td>3,164</td>
<td>0.2</td>
<td>0.3</td>
<td>0.3</td>
<td>6,825</td>
<td>11,675</td>
<td>12,400</td>
</tr>
<tr>
<td>JarScan</td>
<td>884*</td>
<td>274,332</td>
<td>2</td>
<td>12</td>
<td>25</td>
<td>5,840</td>
<td>10,300</td>
<td>10,420</td>
</tr>
<tr>
<td>Jally</td>
<td>2,309*</td>
<td>900,045</td>
<td>21</td>
<td>35</td>
<td>50</td>
<td>12,736</td>
<td>24,272</td>
<td>24,420</td>
</tr>
<tr>
<td>JDictionary</td>
<td>13,423</td>
<td>90,688</td>
<td>16</td>
<td>22</td>
<td>26</td>
<td>24,580</td>
<td>52,240</td>
<td>54,060</td>
</tr>
<tr>
<td>JPassGen</td>
<td>6,673</td>
<td>30164</td>
<td>1.2</td>
<td>5.1</td>
<td>5.5</td>
<td>8,096</td>
<td>15,108</td>
<td>15,128</td>
</tr>
<tr>
<td>Jtidy</td>
<td>23,991</td>
<td>965,964</td>
<td>2</td>
<td>57</td>
<td>148</td>
<td>7,900</td>
<td>55,920</td>
<td>73,972</td>
</tr>
<tr>
<td>JackSum</td>
<td>4,017</td>
<td>297,365</td>
<td>3</td>
<td>11.6</td>
<td>11.8</td>
<td>5,248</td>
<td>12,664</td>
<td>12,896</td>
</tr>
<tr>
<td>IZPress</td>
<td>1,748</td>
<td>1,800,090</td>
<td>23</td>
<td>45</td>
<td>50</td>
<td>13,544</td>
<td>23,672</td>
<td>24,012</td>
</tr>
<tr>
<td>Diff</td>
<td>703</td>
<td>710,125</td>
<td>0.42</td>
<td>16</td>
<td>38</td>
<td>5,264</td>
<td>14,138</td>
<td>18,184</td>
</tr>
<tr>
<td>JavaTar</td>
<td>5,163</td>
<td>366,207</td>
<td>3.5</td>
<td>12.8</td>
<td>17.3</td>
<td>5,912</td>
<td>17,212</td>
<td>17,260</td>
</tr>
<tr>
<td>javap</td>
<td>1,382</td>
<td>11,099</td>
<td>0.75</td>
<td>3.4</td>
<td>3.5</td>
<td>7,026</td>
<td>20,492</td>
<td>20,504</td>
</tr>
<tr>
<td>javac</td>
<td>12,807</td>
<td>2,616,353</td>
<td>1.5</td>
<td>121</td>
<td>203</td>
<td>11,048</td>
<td>119,084</td>
<td>216,212</td>
</tr>
</tbody>
</table>

61
Chapter 8

Implicit information flows

A mechanism that detects implicit flows will likely generate many false positives since it must be entirely or partially based on static analysis. An illegal information leak resulting from an implicit flow is no less serious than one resulting from an explicit flow, but in some circumstances it might be more costly to deal with many false positives than to recover from few illegal leaks. This is why the detection of implicit flows must be made optional in any mechanism that detects insecure information flows.

The \textit{DIFA} algorithm presented in Section 5.1 is capable of detecting all explicit information flows as described in Chapter 1, but its dynamic nature impedes it from detecting any implicit flows. Note that in [13] all flows resulting from control constructs are considered implicit. Our characterization of implicit flow is somewhat different in that when a statement defining the target of a flow actually executes, the flow is considered explicit. Such a flow is detectable by our \textit{DIFA} algorithm. An innovative aspect of our approach is that it includes an optional preliminary static analysis phase that identifies implicit flows and transforms them into explicit ones. The resulting \textit{hybrid} mechanism is therefore capable of detecting both explicit and implicit information flows. A clear advantage of our hybrid mechanism over purely static ones is that it can accurately determine the sources of the flows even if they involve object or array references. Another advantage is that the sensitivity levels of the sources of the flows are configurable and evaluated at runtime.
For the purpose of the aforementioned transformation, our tool’s API provides a method declared as ‘void addFlow(s_id, x_id)’ where s_id identifies a decision statement s and x_id identifies a variable x. addFlow records the influence of s on x by adding the InfoFlow and DynSlice sets last computed for s to those for x. This has the equivalent effect to executing statement x = x in the direct scope of s. The code transformation process involves statically identifying implicit flows (or implicit influences) and then inserting calls to addFlow accordingly.

We first present examples of various Java programming constructs and their corresponding transformations then present a transformation algorithm and discuss its limitations. Finally, we discuss how to address those limitations. Note that in this chapter we assume that an intruder trying to access sensitive information may have knowledge of the code being analyzed.

### 8.1 if-else statements

Consider the code in Figure 14-a where y1 and y2 are binary variables, and send is a method intended to disclose non-sensitive data to the outside world (these assumptions are made throughout this chapter). If line 2 executes (i.e., (y1 == 1) and (x1 == 1) holds) our basic DIFA algorithm will detect the explicit information flow from y1 at line 1 to x1 at line 2 and then to the send statement at line 6. On the other hand, if line 2 does not execute (i.e., (y1 == 0) and (x1 == 0) holds) the basic algorithm will not detect the implicit flow from y1 at line 1 to x1 at line 6, by which an intruder can infer that y1 == 0 after observing that x1 == 0. Other implicit flows will also missed by our algorithm unless the transformation in Figure 14-b is performed. In this transformed code, each inserted addFlow call records an implicit influence from a statement to a
\[ x_1 = x_2 = x_3 = 0; \]
1. if \((y_1 == 1)\) 
   { 
   2. \(x_1 = 1;\) 
   } 
   else 
   { 
   3. if \((y_2 == 1)\) 
       { 
       4. \(x_2 = 1;\) 
             } 
       else 
       { 
       5. \(x_3 = 1;\) 
             } 
   } 
6. send(x1); 
7. send(x2); 
8. send(x3); 

\[ x_1 = x_2 = x_3 = 0; \]
1. if \((y_1 == 1)\) 
   { 
   T1. \(addFlow(1, \text{x}_2_{\text{id}});\) 
   T2. \(addFlow(1, \text{x}_3_{\text{id}});\) 
   2. \(x_1 = 1;\) 
   } 
   else 
   { 
   T3. \(addFlow(1, \text{x}_1_{\text{id}});\) 
   3. if \((y_2 == 1)\) 
       { 
       T4. \(addFlow(3, \text{x}_3_{\text{id}});\) 
       4. \(x_2 = 1;\) 
             } 
       else 
       { 
       T5. \(addFlow(3, \text{x}_2_{\text{id}});\) 
       5. \(x_3 = 1;\) 
             } 
   } 
6. send(x1); 
7. send(x2); 
8. send(x3); 

\textbf{Figure 14} – a) Original code. b) Transformed code.

variable allowing our \textit{DIFA} algorithm to track the resulting implicit flows. Note that an intruder familiar with the code will realize that: for \(x_1\) to stay 0, the condition \((y_1 == 0)\) must hold; for \(x_2\) to stay 0, the condition \(((y_1 == 1) \text{ or } (y_2 == 0))\) must hold; and for \(x_3\) to stay 0, the condition \(((y_1 == 1) \text{ or } (y_2 == 1))\) must hold. The following cases illustrate why the transformation is needed:

1) When \(y_1\) is sensitive, \(y_2\) is known to be 1, and line 4 did not execute, the \texttt{addFlow} call at \(T_1\) prevents line 7 from executing and disclosing that \(x_2 == 0\), from which an intruder could infer that \(y_1 == 1\).
2) When \( y_1 \) is sensitive, \( y_2 \) is known to be 0, and line 5 did not execute, the
\texttt{addFlow} call at \( T_2 \) prevents line 8 from executing and disclosing that \( x_3 == 0 \),
from which an intruder could infer that \( y_1 == 1 \).

3) When \( y_1 \) is sensitive and line 2 did not execute, the \texttt{addFlow} call at \( T_3 \) prevents
line 6 from executing and disclosing that \( x_1 == 0 \), from which an intruder could
infer that \( y_1 == 0 \).

4) When \( y_2 \) is sensitive, \( y_1 \) is known to be 0, and line 5 did not execute, the
\texttt{addFlow} call at \( T_4 \) prevents line 8 from executing and disclosing that \( x_3 == 0 \),
from which an intruder could infer that \( y_2 == 1 \).

5) When \( y_2 \) is sensitive, \( y_1 \) is known to be 0, and line 4 did not execute, the
\texttt{addFlow} call at \( T_5 \) prevents line 7 from executing and disclosing that \( x_2 == 0 \),
from which an intruder could infer that \( y_2 == 0 \).

Note that when \( y_2 \) is sensitive and \( y_1 \) is not, line 7 executes only if \( y_1 == 1 \), disclosing
that \( x_2 == 0 \) and consequently \((y_1 == 1) \) or \((y_2 == 0))\). Nothing is actually disclosed
about the value of \( y_2 \) since the conditional expression is \texttt{true} no matter what \( y_2 \)’s value
is. In addition, the only predicate that uses \( y_2 \) (at line 3) does not actually execute if \( y_1
== 1 \). Therefore it cannot engender explicit or implicit information flows. Similar
arguments apply to when line 8 discloses that \( x_3 == 0 \).

In summary, as a result of the transformation, line 6, which discloses information
about \( y_1 \) only, will never leak sensitive information since it will execute only if \( y_1 \) is not
敏感. Lines 7 and 8, which disclose information about \( y_1 \) and \( y_2 \), will never leak
sensitive information since they will execute only if: a) both \( y_1 \) and \( y_2 \) are not sensitive
or b) \( y_2 \) is sensitive, \( y_1 \) is not, and \( y_1 == 1 \) (see arguments in previous paragraph). Note
1. x1 = x2 = 0;
2. if (y1 == 1) {
3.     x1 = 1;
4.     if (y2 == 1) {
5.         x2 = 1;
6.     }
7.     send(x2);
8. }
9. send(x1);
10. send(x2);

1. x1 = x2 = 0;
2. if (y1 == 1) {
3.     x1 = 1;
4.     if (y2 == 1) {
5.         x2 = 1;
6.     }
7.     addFlow(4, x2_id);
8.     send(x2);
9. }
10. addFlow(2, x1_id)
11. addFlow(2, x2_id);
12. send(x1);
13. send(x2);

Figure 15 – a) Original code. b) Transformed code.

that if neither of y1 or y2 is sensitive; our tool will track the described (implicit) flows but will not signal an illegal flow at any of the send statements.

8.2 if statements

In the absence of an else branch, the false branch of an if statement skips over the body of the statement. This section shows how the transformation of if statements parallels the transformation of if-else statements. Consider the code in Figure 15-a. If line 3 executes (i.e., (y1 == 1) and (x1 == 1) holds) our mechanism will detect the explicit information flow from y1 at line 2 to x1 at line 3 and then to x1 at line 7. On the other hand, if line 3 does not execute (i.e., (y1 == 0) and (x1 == 0) holds) our mechanism will not detect the implicit flow from y1 at line 2 to x1 at line 7, by which an intruder can infer that y1 == 0 after observing that x1 == 0. Similarly, if line 5 executes (i.e., (y1 == 1) and (y2 == 1) and (x2 == 1) holds) our mechanism will detect the explicit information flow from y2 at line 4 to x2 at line 5 and then to x2 at line 6 and line
8, but if line 5 does not execute, our mechanism will not detect the implicit flow from \( y_2 \) at line 4 to \( x_2 \) at line 6 and line 8. Other implicit flows will also be missed by our algorithm unless the transformation in Figure 15-b is performed. In this transformed code, each inserted \texttt{addFlow} call records an implicit influence from a statement to a variable allowing our \textit{DIFA} algorithm to track the resulting implicit flows. Note that an intruder familiar with the code will realize that: for \( x_1 \) to stay 0, the condition \((y_1 == 0)\) must hold; and for \( x_2 \) to stay 0, the condition \(((y_1 == 0) \text{ or } (y_2 == 0))\) must hold. The following cases illustrate why the transformation is needed:

1) When \( y_2 \) is sensitive and line 5 did not execute, the \texttt{addFlow} call at T1 prevents line 6 from executing and disclosing that \( x_2 == 0 \), from which an intruder could infer that \( y_2 == 0 \). Note that at line 6, \textit{InfoFlow} of \( x_2 \) contains \( y_2 \) as a result of T1 recording the implicit influence of line 4 on \( x_2 \).

2) When \( y_1 \) is sensitive and line 3 did not execute, the \texttt{addFlow} call at T2 prevents line 7 from executing and disclosing that \( x_1 == 0 \), from which an intruder could infer that \( y_1 == 0 \). Note that at line 7, \textit{InfoFlow} of \( x_1 \) contains \( y_1 \) as a result of T2 recording the implicit influence of line 2 on \( x_1 \).

3) When \( y_1 \) is sensitive, \( y_2 \) is known to be 1, and line 5 did not execute, the \texttt{addFlow} call at T3 prevents line 8 from executing and disclosing that \( x_2 == 0 \), from which an intruder could infer that \( y_1 == 0 \). Note that at line 8, \textit{InfoFlow} of \( x_2 \) contains \( y_1 \) as a result of T3 recording the implicit influence of line 2 on \( x_2 \).

Note that when \( y_2 \) is sensitive and \( y_1 \) is not, line 8 executes only if \( y_1 == 0 \), disclosing that \( x_2 == 0 \) and consequently \(((y_1 == 0) \text{ or } (y_2 == 0))\). Nothing is actually disclosed about the value of \( y_2 \) since the conditional expression is \texttt{true} no matter what \( y_2 \)'s value
is. In addition, the only predicate that uses \( y_2 \) (at line 4) does not actually execute if \( y_1 == 0 \). Therefore it cannot engender explicit or implicit information flows.

In summary, as a result of the transformation, line 6, which discloses information about \( y_1 \) and \( y_2 \), will never leak sensitive information since it will execute only if \( y_1 \) and \( y_2 \) are both not sensitive. Line 7, which discloses information about \( y_1 \) only, will never leak sensitive information since it will execute only if \( y_1 \) is not sensitive. Line 8, which discloses information about \( y_1 \) only (see arguments above), will never leak sensitive information since it will execute only if \( y_1 \) is not sensitive.

### 8.3 Other control statements

The basic idea behind the transformation applied to the if and if-else statements also applies to any control construct, that is, given a decision statement \( p \), the transformation inserts at the start of each branch of \( p \) `addFlow` calls reflecting the explicit influences due to \( p \) that the execution might have encountered if the alternative branches were taken.

The first column of Figures 16 and 17 show Java code involving various control statements, the second column shows the corresponding transformations. Arguments similar to the ones presented in Section 8.1 and 8.2 can be used to justify the proposed transformations.
<table>
<thead>
<tr>
<th>a)</th>
<th>b)</th>
</tr>
</thead>
</table>
| 1. if (y1 == 1)  
   {  
   2. x1 = 1;  
   }  
   else  
   {  
   3. x2 = 1;  
   } | 1. if (y1 == 1)  
   {  
   2. addFlow(1, x2_id);  
   x1 = 1;  
   }  
   else  
   {  
   3. addFlow(1, x1_id);  
   x2 = 1;  
   } |
| c) | d) |
| 1. switch(y1)  
   {  
   case 0:  
   2. x1 = 1;  
   break;  
   case 1:  
   3. x2 = 1;  
   break;  
   case 2:  
   4. x3 = 1;  
   break;  
   default:  
   print("invalid case");  
   } | 1. switch(y1)  
   {  
   case 0:  
   addFlow(1, x2_id);  
   addFlow(1, x3_id);  
   x1 = 1;  
   break;  
   case 1:  
   addFlow(1, x1_id);  
   addFlow(1, x3_id);  
   x2 = 1;  
   break;  
   case 2:  
   addFlow(1, x1_id);  
   addFlow(1, x2_id);  
   x3 = 1;  
   break;  
   default:  
   addFlow(1, x1_id);  
   addFlow(1, x2_id);  
   addFlow(1, x3_id);  
   print("invalid case");  
   } |

Figure 16 – Original and transformed Java code involving various control statements
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. switch(y1) {</td>
<td>1. switch(y1) {</td>
<td>1. switch(y1) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>case 0:</td>
</tr>
<tr>
<td>2. x1 = 1;</td>
<td>2. x1 = 1;</td>
<td>addFlow(1, x2_id);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>break;</td>
</tr>
<tr>
<td>case 1:</td>
<td>case 1:</td>
<td>addFlow(1, x3_id);</td>
</tr>
<tr>
<td>3. x2 = 1;</td>
<td>3. x2 = 1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>break;</td>
</tr>
<tr>
<td>case 2:</td>
<td>case 2:</td>
<td></td>
</tr>
<tr>
<td>4. x3 = 1;</td>
<td>4. x3 = 1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>break;</td>
</tr>
<tr>
<td>default:</td>
<td>default:</td>
<td>addFlow(1, x1_id);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addFlow(1, x3_id);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addFlow(1, x3_id);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>a)</td>
<td>b)</td>
<td></td>
</tr>
<tr>
<td>1. x1 = x2 = 0;</td>
<td>1. x1 = x2 = 0;</td>
<td>1. x1 = x2 = 0;</td>
</tr>
<tr>
<td>2. for (int i = 0; i &lt; y1; i++) {</td>
<td>2. for (int i = 0; i &lt; y1; i++) {</td>
<td>2. for (int i = 0; i &lt; y1; i++) {</td>
</tr>
<tr>
<td>3. x1++;</td>
<td>3. x1++;</td>
<td>3. x1++;</td>
</tr>
<tr>
<td>4. for (int j = 0; j &lt; y2; j++) {</td>
<td>4. for (int j = 0; j &lt; y2; j++) {</td>
<td>4. for (int j = 0; j &lt; y2; j++) {</td>
</tr>
<tr>
<td>5. x2++;</td>
<td>5. x2++;</td>
<td>5. x2++;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>6. send(x1);</td>
<td>6. send(x1);</td>
<td>addFlow(4, x1_id);</td>
</tr>
<tr>
<td>7. send(x2);</td>
<td>7. send(x2);</td>
<td>addFlow(4, x2_id);</td>
</tr>
<tr>
<td>c)</td>
<td>d)</td>
<td></td>
</tr>
<tr>
<td>do {</td>
<td>do {</td>
<td>do {</td>
</tr>
<tr>
<td>2. x1++;</td>
<td>2. x1++;</td>
<td>2. x1++;</td>
</tr>
<tr>
<td>3. y1++;</td>
<td>3. y1++;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>4. while (y1 &lt; 10)</td>
<td>4. while (y1 &lt; 10)</td>
<td>addFlow(4, x1_id);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addFlow(4, x1_id);</td>
</tr>
<tr>
<td>5. send(x1);</td>
<td>5. send(x1);</td>
<td>addFlow(4, y1_id);</td>
</tr>
<tr>
<td>e)</td>
<td>f)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 17** – Original and transformed Java code involving various control statements
8.4 Transformation algorithm

This section presents an algorithm that identifies potential implicit flows and then performs the appropriate code transformation to enable our DIFA algorithm to monitor both implicit and explicit flows. This algorithm handles only implicit flows whose targets are simple variables, i.e., local or static variables that are not object references.

The algorithm shown in Figure 18 applies to an individual method and requires that its control flow graph $G$ and immediate postdominators have been computed beforehand.

Given a decision vertex $p$, the algorithm inserts at the start of each branch $s$ of $p$ addFlow calls reflecting the explicit influences (explicit control flows) due to $p$ that the execution might have encountered if the alternative branches were taken. For example in the true branch of the if-else statement of Figure 16-a, it inserts a call that reflects the influence of line 1 on $x_2$, which is defined in the false branch, whereas in the false branch it inserts a call that reflects the influence of line 1 on $x_1$, which is defined in the true branch. Note that when a successor $s$ of $p$ is also ipd($p$), and $s$ has multiple predecessors, the algorithm inserts the required addFlow calls in a new vertex placed between $p$ and $s$, as opposed to inserting them in $s$. This case might apply to code that involves control statements that have a false branch that skips over the body of the statement, such as if statements, switch statements without a default case, and loops.

Note that in the transformations shown so far, only the one transforming the switch statement of Figure 17-a involved adding a new vertex, it is reflected in the added default case shown in Figure 17-b. The next example clarifies the need for adding the aforementioned new vertices.
procedure transformImplicitToExplicit(G)

begin

for each decision vertex p in G begin
    for each s \in successors(p) begin
        if s = ipd(p) and Cardinality(predecessors(s)) > 1
            insertLocation = createVertex(p, s, G)
        else
            insertLocation = s

        for each s' \in successors(p) - \{s\}
            insertCalls(insertLocation, p, Defs(reachable(s', ipd(p), G - \{s\})))
    end
end

end

procedure insertCalls(s, p, targets)
/* Inserts, at the beginning of (basic block) s, calls to addFlow(p_id, t) for all t in targets, where p_id is the identifier (line #) of the only decision statement in p */

function reachable(start, end, G')
/* Returns all nodes in G' - \{end\} reachable from start (inclusive) */

function Defs(G')
/* Returns the set of variables defined in sub-graph G' */

function createVertex(p, s, G)
/* Creates a new vertex between p and s then returns it */

**Figure 18** – Transformation algorithm

Consider the code in Figure 19-a and its control flow graph in Figure 20; the two predicate statements on line 2 and line 4 share the same ipd, i.e., ipd(v_1) = ipd(v_2) = v_4. If the algorithm ignores this fact and does not place a new vertex between v_1 and v_4 and another one between v_2 and v_4, i.e., keeps the original control flow graph, the
1. \(x_1 = x_2 = 0;\)
2. if \((y_1 == 1)\) 
   
   \[
   \begin{align*}
   3. & \quad x_1 = 1; \\
   4. & \quad \text{if } (y_2 == 1) \\
   5. & \quad \quad x_2 = 1; \\
   \end{align*}
   \]
3. print("start of ipd");

<table>
<thead>
<tr>
<th>a)</th>
<th>b)</th>
<th>c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (x_1 = x_2 = 0;)</td>
<td>1. (x_1 = x_2 = 0;)</td>
<td>1. (x_1 = x_2 = 0;)</td>
</tr>
<tr>
<td>2. if ((y_1 == 1))</td>
<td>2. if ((y_1 == 1))</td>
<td>2. if ((y_1 == 1))</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>3. (x_1 = 1;)</td>
<td>3. (x_1 = 1;)</td>
<td>3. (x_1 = 1;)</td>
</tr>
<tr>
<td>4. if ((y_2 == 1))</td>
<td>4. if ((y_2 == 1))</td>
<td>4. if ((y_2 == 1))</td>
</tr>
<tr>
<td>5. (x_2 = 1;)</td>
<td>5. (x_2 = 1;)</td>
<td>5. (x_2 = 1;)</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>T1. addFlow(4, x2_id);</td>
<td>T1. addFlow(4, x2_id);</td>
<td>T1. addFlow(4, x2_id);</td>
</tr>
<tr>
<td>T2. addFlow(2, x1_id)</td>
<td>T2. addFlow(2, x1_id)</td>
<td>T2. addFlow(2, x1_id)</td>
</tr>
<tr>
<td>T3. addFlow(2, x2_id);</td>
<td>T3. addFlow(2, x2_id);</td>
<td>T3. addFlow(2, x2_id);</td>
</tr>
<tr>
<td>6. print(&quot;start of ipd&quot;);</td>
<td>6. print(&quot;start of ipd&quot;);</td>
<td>6. print(&quot;start of ipd&quot;);</td>
</tr>
</tbody>
</table>

**Figure 19** – a) Original code. b) Transformed code without adding new vertices. c) Transformed code with adding new vertices.

**Figure 20** – Control flow graph of the code in Figure 19-a and 19-b.

**Figure 21** – Control flow graph of the code in Figure 19-c.
transformation in Figure 19-b results. In this transformed code T1 records the influence of line 4 on \texttt{x}_2 even if line 4 does not execute, this is obviously not a desirable behavior. Figure 19-c shows the desired transformation, and Figure 21 shows the corresponding modified control flow graph, which contains two additional vertices \( v_5 \) and \( v_6 \).

When a successor \( s \) of \( p \) is also \( \text{ipd}(p) \), but \( s \) has a single predecessor, the algorithm maintains its default behavior and inserts the required \texttt{addFlow} calls in \( s \) itself (see Figures 15-a, 17-c and 17-e). Note that in such cases the resulting \texttt{addFlow} calls effectively become no-ops if the body of the statement does execute, since the influence was explicit and therefore already recorded.

In addition to enabling our \textit{DIFA} algorithm to detect implicit flows, the transformation algorithm enables our dynamic slicing algorithm to optionally identify statements that \textit{implicitly} influence a given variable. This feature is due to the fact that when recording the influence of a statement \( s \) on a variable \( x \), the \texttt{addFlow} method adds \texttt{DynSlice} of \( s \) to \texttt{DynSlice} of \( x \).

During debugging it is valuable to know whether a given information flow from a statement \( s \) to a variable \( x \) occurred implicitly or explicitly. To convey whether an implicit flow actually occurred, the profiler will add the \texttt{addFlow} call statement to the slice of \( x \) only when the flow was implicit. This is decided at runtime, at the start of the \texttt{addFlow} method, by checking whether the \texttt{InfoFlow} of \( x \) already comprises the \texttt{InfoFlow} of \( s \), i.e., whether \( x \) was already explicitly influenced by \( s \).

We now discuss the limitations of our transformation algorithm and how they can be addressed. The second parameter of the \texttt{addFlow} method identifies the target of the flow. Our current transformation algorithm handles implicit flows involving only targets
that are simple variables, since these can be easily identified statically. Points-to-analysis [29] could be used to enable our algorithm to support implicit flows involving targets that are object references. There are tools that can perform points-to analysis on java programs, such as Soot [71], but these are based on their own intermediate representation rather than Java byte code, making their integration with our tool infeasible.

The current transformation algorithm does not handle inter-procedural implicit flows, i.e., flows where the targets are defined in called functions. Providing such support is straightforward for static functions but not for non-static ones. Given a statement calling a non-static function, the algorithm must determine, with some level of accuracy, the functions that could actually be called. Type inference [1], which is related to points-to-analysis, could be used to increase the accuracy of this identification.

Finally, array region analysis [82], which summarizes the sub-region of an array that is affected by statements or loops, could enable our algorithm to identify with more accuracy implicit flow targets that are array references.

Sections 8.5, 8.6 and 8.7 further discuss the above subjects and Appendix B supports the correctness of our transformation algorithm.

8.5 Implicit flows with non-simple targets

Points-to-analysis [29][49][71] is a compiler optimization research area whose goal is to statically find information about the set of objects that a given object reference potentially points to. This section shows example code where the target of an implicit flow involves an object reference then shows how points-to-analysis can be exploited to provide an appropriate transformation. In the code shown in Figure 22-a, the object reference returned by the call to getA() on line 6 can potentially point to one of three
different instances of A, namely \texttt{a1}, \texttt{a2} and \texttt{a4}. During our transformation, we are not interested in identifying the types of those objects but in identifying the objects themselves. We recognize that in the static context, we can identify an object by the location of the \texttt{new} expression that potentially defines it, e.g. \texttt{a1} is identified by line 1, \texttt{a2} by line 2 and \texttt{a4} by line 4. We count on points-to-analysis to provide such information, i.e., for each object reference the locations of the \texttt{new} expressions that potentially define it. Figure 22-b shows the transformation of the code in Figure 22-a based on the above observation. Information identifying all potential targets is passed to a modified version of the \texttt{addFlow} method. It includes the name of the influenced field and the locations of the \texttt{new} expressions associated with the object reference. At runtime, the \texttt{addFlow} method retrieves the corresponding object instances and records the influence of the source statement on their respective ‘\texttt{data}’ fields. Clearly, the more active objects are found the more conservative the analysis is, since only one of the identified targets was actually involved in the implicit flow. Note that in the case when the targets of the implicit flows involve object references that are array elements, the transformation uses a different scheme than what’s described above, as we discuss in Section 8.7.
class A
{
    int data;
    ...
}
public class Test
{
    public static int y;
    public static A a1, a2, a3, a4;
    static {
        1.   a1 = new A(...);
        2.   a2 = new A(...);
        3.   a3 = new A(...);
        4.   a4 = new A(...);
    }
    public static A getA(int select)
    {
        switch(select)
        {
            case 1:
                return a1;
            case 2:
                return a2;
            default:
                return a4;
        }
    }
    public static void foo(int select)
    {
        5.    if (y == 1)
        {
            6.        getA(select).data = 1;
        }
    }
}

Figure 22 – a) Original code. b) Transformed code.
8.6 Inter-procedural implicit flows

Inter-procedural implicit flows are flows whose targets are defined in called functions. Figure 23-a shows code involving a statement calling a static function foo(). When y1 is sensitive and (y1 != 1) holds, this code leaks sensitive information about y1 through x1. This becomes clear if foo() is inlined at line 4. A transformation that would enable our DIFA algorithm to detect such implicit flow is shown Figure 23-b. Static analysis can determine with certainty what function will be called at line 4 since foo() is static. If the statement at line 4 was calling a non-static function, our static transformation algorithm must first determine, with some level of accuracy, the set of functions that could actually be called. The code in Figure 24 illustrates the problem. If line 3 of Figure 24-b does not execute (i.e., (y1 != 1) holds) our DIFA algorithm will not detect the implicit flow from y1 at line 2 to x1 at line 4 or x3 at line 6. Note that no implicit flow occurs from y1 to x2 since the foo() method of class C could never be called. A transformation that would enable our algorithm to detect such implicit flows is shown Figure 24-c. Such transformation requires the algorithm to accurately identify all the foo() methods that potentially could be called, i.e., recognize that only the foo() methods in class B and class D can potentially be called. Type inference [1][2][3][58][59], which is related to points-to-analysis, is a compiler optimization research area that deals with such issue. The goal is to inline the identified methods, if possible, to avoid the overhead of function calls. The techniques developed by this research area can be used to increase the accuracy of our method identification, thus leading to a less conservative information flow analysis.
| static int x1;  
|   static void foo( )  
|   {  
|       int x2 = 1;  
|   1. x1 = 1;  
|   }  
| 3. if (y1 == 1)  
|  {  
|  4. foo( );  
|  }  
| 5. print(x1);  |

| static int x1;  
|   static void foo( )  
|   {  
|       int x2 = 1;  
|   1. x1 = 1;  
|   }  
| 3. if (y1 == 1)  
|  {  
|  4. foo( );  
|  }  
| T1. addFlow(3, x1_id);  
| 5. print(x1); |

**Figure 23** – a) Original code. b) Transformed code.
```java
class A {
    static int x1;
    static int x2;
    static int x3;
}

class B extends A {
    void foo() {
        x1 = 1;
    }
}

class C extends A {
    void foo() {
        x2 = 1;
    }
}

class D extends A {
    void foo() {
        x3 = 1;
    }
}

1. A obj = (...) ? new B() : new D();
2. if (y1 == 1) {
3.     obj.foo();
4. }
5. print(x1);
6. print(x2);
7. print(x3);

b) 1. A obj = (...) ? new B() : new D();
2. if (y1 == 1) {
3.     obj.foo();
4. }    
   addFlow(1, x1_id);
   addFlow(1, x3_id);
5. print(x1);
6. print(x2);
7. print(x3);

c) Figure 24 – a) Class hierarchy. b) Original code. c) Transformed code.
8.7 Arrays and array elements

Like any variable, array elements can become targets of explicit or implicit information flows. In some cases, the array itself is the target of a flow. In Figure 25-c, there is an explicit data flow from $y$ to $a[0]$ that our basic \textit{DIFA} algorithm is capable of detecting. When $y \neq 1$ in Figure 25-a, our \textit{DIFA} algorithm will not detect the implicit flow from $y$ to $a[i]$, therefore a transformation such as the one in Figure 25-b is required. Note that since static analysis is unable to determine the specific value of $i$, T1 records the influence of line 1 on the array itself, i.e., on all the elements of the array.

In Figure 25-d our \textit{DIFA} algorithm detects the (explicit) flow from $y$ to $a[y]$. Therefore, when $y$ is sensitive, the \texttt{print} statement at line 2 will not execute if $y == 0$, but will execute otherwise disclosing that $a[0] == 0$, from which an intruder could infer that $y \neq 0$. This leak of sensitive information occurs as a result of implicit flows at line 1 from $y$ to all the elements of array $a$ other than $a[y]$. In summary, the execution of line 1 can cause an explicit flow from $y$ to $a[y]$ and implicit flows from $y$ to all the elements of array $a$ other than $a[y]$. In order to detect both the explicit and implicit flows, the \textit{DIFA} algorithm must record the flow from $y$ to the array $a$ itself. Note that no static transformation is needed to enable such behavior.

Figure 26-a shows code involving loops, array references and a sensitive variable $y$. When $y == 0$, this code is secure since the \texttt{print} statement will never execute as a result of the \textit{DIFA} algorithm detecting all the explicit flows from $y$ to all the elements of $a$. When $y \neq 0$, this code leaks sensitive information about $y$ due to undetected implicit flows. For example, if $y == 1$, line 5 discloses that $a[0] == 0$, from which an intruder
could infer that $y \neq 0$. Figure 26-b shows a transformation that would enable our DIFA algorithm to detect the insecure implicit flows. The addFlow call at T1 records the influence of line 2 on every element of array $a$. This might register flows that have already been explicitly registered. Note that we are only interested in recording the influence of line 2 on the elements of $a$ that were not defined at line 3, i.e., the ones that are implicitly influenced by line 2. Array region analysis [82] is a compiler optimization technique that summarizes the sub-region of an array that is affected by statements or loops. Consider the loop:

```c
for (int i = 1; i <= N; i++)
{
    b[i] = c[2:N-i] + 1;
}
```

The goal is to recognize that the region of array $b$ that is defined is $b[1:N]$ and of array $c$ that is used is $c[N:2N-1]$. The outcome of conducting array region analysis on our code recognizes that lines 2 and 3 define $a[y:9]$. We could use this information to refine the conservative transformation of Figure 26-b to yield the transformation of Figure 26-c.

Finally, Figure 27 shows transformed code where the targets of the flows involve array elements referencing objects. Note that such transformations are not based on points-to-analysis, but in some cases require array region analysis, as in Figure 27-c. As shown, a modified version of the addFlow method is used.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1. if (y == 1)  
  {  
  2. a[i] = 1;  
  }  
  3. print(a[i]); | 1. if (y == 1)  
  {  
  2. a[i] = 1;  
  }  
  T1. addFlow(1, a_id);  
  3. print(a[i]); |   |
| a) | b) |   |
| 1. a[0] = y;  
  2. print(a[0]); | 1. a[y] = 1;  
  2. print(a[0]); | c) |
|   |   | d) |

**Figure 25** – a) Original code. b) Transformed code. c) Explicit flow. d) Explicit and implicit flow.
1. int a[] = new int[10];
2. for (int i = y; i < 10; i++)
   {
3.        a[i] = 1;
   }
4. for (int i = 0; i < 10; i++)
   {
5.    print(a[i]);
   }

<table>
<thead>
<tr>
<th>a)</th>
<th>b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. int a[] = new int[10];</td>
<td>1. int a[] = new int[10];</td>
</tr>
</tbody>
</table>
| 2. for (int i = y; i < 10; i++)
   {
3.        a[i] = 1;
   } | 2. for (int i = y; i < 10; i++)
   {
3.        a[i] = 1;
   } |
| T1. addFlow(2, a_id); | |
| 4. for (int i = 0; i < 10; i++)
   {
5.    print(a[i]);
   } | 4. for (int i = 0; i < 10; i++)
   {
5.    print(a[i]);
   } |

<table>
<thead>
<tr>
<th>c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. int a[] = new int[10];</td>
</tr>
</tbody>
</table>
| 2. for (int i = y; i < 10; i++)
   {
3.        a[i] = 1;
   } |
| for all k belonging to [0, y-1] (i.e. complement of [y, 9]) |
| addFlow(2, a[k_id]); |
| 4. for (int i = 0; i < 10; i++)
   {
5.    print(a[i]);
   } |

**Figure 26** - a) Original code. b) Transformed code. c) Transformed code based on array region analysis.
<table>
<thead>
<tr>
<th>a)</th>
<th>b)</th>
</tr>
</thead>
</table>
| 1. if (y == 1)  
   {  
     a[i].data = 1;  
   }  
T1. addFlow(1, a_id, 'data');  
3. print(a[i].data); | 1. A a [] = new A[10];  
2. for (int i = y; i < 10; i++)  
   {  
     a[i].data = 1;  
   }  
T1. addFlow(2, a_id, 'data');  
4. for (int i = 0; i < 10; i++)  
   {  
     print(a[i].data);  
   } |

1. for all k belonging to [0, y-1] (i.e. complement of [y, 9])  
   addFlow(2, a[k]_id, 'data');

4. for (int i = 0; i < 10; i++)  
   {  
     print(a[i].data');  
   }

c) **Figure 27** - Transformed code where the targets of the flows involve array elements referencing objects.
Chapter 9

Test-case filtering

Test-case filtering is concerned with selecting a subset of a test suite that is capable of detecting most or all of the defects detected by the original test suite. Leon and Podgurski presented a comparison of coverage-based and distribution-based (OBT) techniques for filtering test cases [47]. Their work presented an empirical comparison of four different techniques for filtering test suites: test suite minimization [83], prioritization by additional coverage [17][18], cluster filtering with one-per-cluster sampling [15], and failure pursuit sampling [16]. Their experiments involved three subject programs: the GCC C compiler [26], the Jikes Java compiler [38] and the javac Java compiler [34]. The used the following three profiles: method calls, basic blocks, and control flow edges between basic blocks. This dissertation augments their work by:

- Using in addition to javac two other subject programs: the Xerces XML parser [84] and the JTidy HTML syntax checker and pretty printer [40].
- Using profiles that capture relatively more complex interactions between the components of an executing program: information flow profiles and slice profiles.

Note that this chapter also presents profiles based on techniques contributed by Leon but not presented in [47], namely, data flow profiles and method pair profiles.

In this chapter we will briefly describe the test-case filtering techniques used in our experiments. Describe our subject programs and test suites. Finally, we will present our
empirical results and discuss the comparative defect detection efficiency of the used techniques and profiles types.

### 9.1 Coverage-based techniques

Coverage-based testing is a type of structural testing that measures the quality of a test suite by how fully it exercises (covers) specific features of a program. For example, for a test suite to achieve 100% statement (or basic block) coverage, each statement in the program has to be executed by at least one test case from this suite. Other program features used by coverage-based testing include methods, branches and def-use pairs.

The simplest type of coverage-based test-case filtering is *test suite minimization* [83], it involves selecting the smallest subset of a test suite that covers as many program features as the original test suite does. As in [47] we will refer to this as *basic coverage maximization*. An extension to this technique is prioritization by additional coverage [17][18], it involves conducting basic coverage maximization repeatedly on the set of test cases that have not yet been selected. [47] refers to this as *repeated coverage maximization*. Repeated coverage maximization could be employed in case basic coverage maximization selected only a few tests and more is desired. Note that our experiments did not involve repeated coverage maximization.

### 9.2 OBT techniques

Test-case filtering techniques in OBT involve partitioning a set of tests into clusters based on the dissimilarity of their multivariate profiles then selecting samples (test cases) from the resulting clusters. The cluster analysis phase requires the choice of a
dissimilarity metric, such as $n$-dimensional Euclidian distance, and the sampling phase requires a sampling method, such as one-per-cluster sampling (OPC) [15] or failure-pursuit sampling (FP) [16]. OPC sampling randomly selects a single test from each cluster whereas FP sampling selects the $k$ nearest neighbors of any failures found by auditing the subset of tests initially selected through OPC sampling. If any additional failures are found, each of their $k$ nearest neighbors is selected, and so on, until no additional failures are found. Note that FP sampling can cross cluster boundaries.

9.3 Subject Programs

In our experiments we used three Java programs: the javac Java compiler version 1.3.1_02-b02 [34], the Xerces XML parser version 2.1 [84] and the JTidy HTML syntax checker and pretty printer version 3 [40].

javac was tested with the Jacks test suite [32], which tests the compliance with the Java Language Specification [33]. The Jacks test suite comprises 3,140 tests among which 223 caused javac to fail.

Xerces was tested by using part of the XML Conformance Test Suite (XML TS) [85], which provides a set of metrics for determining conformance to the W3C XML Recommendation. There are 2000 tests in the XML TS contributed by several organizations such as Sun and IBM. We used 1663 tests in our experiments resulting in 64 failures. Note that we chose to exclude 333 tests because it was difficult to determine with certainty whether those tests were expected to pass or fail and we excluded 4 other tests because their executions ran out of memory.
JTidy was tested using 1000 files downloaded from the Google Groups [27] using a web crawler. Out of the 1000, 5 were XML files and the rest were HTML files. There were 47 failures out of the 1000 tests.

The sizes (in number of lines) of the subject programs were 12,807 for javac, 52,528 for Xerces and 9,153 for Jtidy. We believe that in the case of Xerces, only a fraction of the functionality was exercised, since the application was configured to only check the syntax and not the semantics of the input XML files, i.e., simply check whether the files were well-formed.

The defects causing the failures were manually investigated and the failures were classified into groups believed to have been caused by the same defect. For javac, 67 distinct defects were believed to have caused the 223 failures. For Xerces, 6 distinct defects were believed to have caused the 64 failures. For JTidy, 5 distinct defects were believed to have caused the 47 failures.

The DIFA tool was configured to generate information flow and slice profiles for all three programs. Note that when executing under such configuration the memory and CPU usage of the DIFA tool increases tremendously since all flows and slices that occur at a given statement need to be saved and tracked and not just the latest ones. In addition, a specialized profiler was built [47] using BCEL [8] to generate the remaining types of profiles, namely, method calls, method call pairs, basic blocks, basic-block edges and def-use pairs. The profiles contain the following information:

- Method calls or MC: Number of times each method was executed.
- Method call pairs or MCP: Number of times each method A calls another method B, for every combination of A and B for which this count is nonzero.
Table 5 - Unique execution counts for the various types of profiles.

*Combines MC, MCP, BB, BBE and DUP

<table>
<thead>
<tr>
<th></th>
<th>MC</th>
<th>MCP</th>
<th>BB</th>
<th>BBE</th>
<th>DUP</th>
<th>*All</th>
<th>IF-NIP</th>
<th>Slice-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac</td>
<td>1,022</td>
<td>2,123</td>
<td>3,655</td>
<td>4,307</td>
<td>9,620</td>
<td>11,315</td>
<td>66,829</td>
<td>-</td>
</tr>
<tr>
<td>Xerces</td>
<td>361</td>
<td>690</td>
<td>1,725</td>
<td>1,982</td>
<td>3,812</td>
<td>4,519</td>
<td>6,547</td>
<td>79,134</td>
</tr>
<tr>
<td>JTidy</td>
<td>209</td>
<td>517</td>
<td>1,698</td>
<td>2,103</td>
<td>5,201</td>
<td>6,080</td>
<td>12,300</td>
<td>-</td>
</tr>
</tbody>
</table>

- Basic-blocks or BB: Number of times a basic block was executed. This is equivalent to a statement profile.
- Basic-block edges or BBE: Number of times control flows between basic-block B1 to basic-block B2, for every combination of B1 and B2 for which this count is nonzero.
- Def-use pairs or DUP: Number of times a statement U uses a variable defined by statement D, for each combination of D and U for which the count is nonzero.
- Information flow with inter-procedural control dependence (InterProcCD) turned off or IF-NIP: Number of times an object A influenced object B (A ∈ InfoFlow(B)), for each combination of A and B for which this count is nonzero.
- Slicing with inter-procedural control dependence (InterProcCD) turned on or Slice-IP: Number of times a statement A appears in a slice computed for statement B (A ∈ DynSlice(B)), for each combination of A and B for which this count is nonzero.

Table 5 shows for each pair of program and profile type the unique execution counts that were encountered while running the test suites. For example, there were
79,134 different combinations of statements making up the Slice-IP profiles for Xerces, where each combination is made up of two statements $s_1$ and $s_2$, such that at least one slice computed at $s_2$ contained $s_1$. Also, there were 6,547 different combinations of information flows making up the IF-NIP profiles for Xerces; where each combination is made up of two objects identifiers $oid_1$ and $oid_2$, such that $oid_1$ influenced $oid_2$ at least once during the execution of the test suite. Note that the data in Table 5 was arrived at following the removal of redundant execution counts (see [47]). Also, the column titled ‘All’ shows the combined counts of MC, MCP, BB, BBE and DUP. Due to redundancy, this combined count is less than the sum of its components counts. For example, the first basic block of a function will be executed every time the function is called, meaning that each count of the MC profile matches one count of the BB profile, and they will be represented by only one count in the combined profile.

As expected, Table 5 shows that using a finer granularity profile leads to a higher number of unique execution counts.

9.4 Basic coverage maximization results and discussions

The results of conducting basic coverage maximization are shown Table 6. They were obtained by running the greedy selection algorithm, described in [47], which attempts to find the minimal test set that covers all of the features covered by the original test suite. For example, in the case of Xerces/Slice-IP, the algorithm finds that no more than 20.93% of the original test suite is needed to exercise all of the slices exercised by the original test suite, and the selected 20.93% tests cover 74.86% of all the defects covered by the original test suite. Note that the data shown in each row of Table 6 is the
<table>
<thead>
<tr>
<th></th>
<th>Profile Type</th>
<th>Selected</th>
<th>%</th>
<th>Failure s</th>
<th>%</th>
<th>Defects</th>
<th>%</th>
<th>Selected/Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MCP</td>
<td>164.345</td>
<td>5.23</td>
<td>15.151</td>
<td>6.77</td>
<td>15.145</td>
<td>22.6</td>
<td>10.85144</td>
</tr>
<tr>
<td></td>
<td>BB</td>
<td>245.076</td>
<td>7.8</td>
<td>20.07</td>
<td>8.96</td>
<td>19.1</td>
<td>28.5</td>
<td>12.8312</td>
</tr>
<tr>
<td></td>
<td>BBE</td>
<td>315.765</td>
<td>10.05</td>
<td>25.227</td>
<td>11.3</td>
<td>21.928</td>
<td>32.8</td>
<td>14.40008</td>
</tr>
<tr>
<td></td>
<td>DUP</td>
<td>572.137</td>
<td>18.22</td>
<td>53.017</td>
<td>23.7</td>
<td>41.915</td>
<td>62.5</td>
<td>13.64993</td>
</tr>
<tr>
<td>*All</td>
<td></td>
<td>606.31</td>
<td>19.3</td>
<td>55.149</td>
<td>24.7</td>
<td>42.392</td>
<td>63.2</td>
<td>14.30246</td>
</tr>
<tr>
<td>IF-NIP</td>
<td></td>
<td>589.591</td>
<td>18.77</td>
<td>57.515</td>
<td>25.8</td>
<td>45.628</td>
<td>68.1</td>
<td>12.92169</td>
</tr>
<tr>
<td>Xerces</td>
<td>MC</td>
<td>15.242</td>
<td>0.91</td>
<td>0.037</td>
<td>0.57</td>
<td>0.037</td>
<td>0.61</td>
<td>411.9459</td>
</tr>
<tr>
<td></td>
<td>MCP</td>
<td>59.56</td>
<td>3.58</td>
<td>0.262</td>
<td>0.409</td>
<td>0.262</td>
<td>4.36</td>
<td>227.3282</td>
</tr>
<tr>
<td></td>
<td>BB</td>
<td>161.86</td>
<td>9.74</td>
<td>1.345</td>
<td>2.1</td>
<td>1.345</td>
<td>22.41</td>
<td>120.342</td>
</tr>
<tr>
<td></td>
<td>BBE</td>
<td>198.213</td>
<td>11.91</td>
<td>1.858</td>
<td>2.9</td>
<td>1.858</td>
<td>30.96</td>
<td>106.6808</td>
</tr>
<tr>
<td></td>
<td>DUP</td>
<td>262.317</td>
<td>15.77</td>
<td>4.269</td>
<td>6.67</td>
<td>3.269</td>
<td>54.48</td>
<td>80.24381</td>
</tr>
<tr>
<td>*All</td>
<td></td>
<td>279.235</td>
<td>16.78</td>
<td>4.139</td>
<td>6.46</td>
<td>3.139</td>
<td>52.31</td>
<td>88.95667</td>
</tr>
<tr>
<td>IF-NIP</td>
<td></td>
<td>112.312</td>
<td>6.75</td>
<td>1.445</td>
<td>2.25</td>
<td>1.445</td>
<td>24.08</td>
<td>77.72457</td>
</tr>
<tr>
<td>Slice-IP</td>
<td></td>
<td>348.257</td>
<td>20.93</td>
<td>5.492</td>
<td>8.58</td>
<td>4.492</td>
<td>74.86</td>
<td>77.52827</td>
</tr>
<tr>
<td>JTidy</td>
<td>MC</td>
<td>11.38</td>
<td>1.138</td>
<td>2</td>
<td>0.65</td>
<td>2.88</td>
<td>57.6</td>
<td>3.951389</td>
</tr>
<tr>
<td></td>
<td>MCP</td>
<td>30.87</td>
<td>3.087</td>
<td>4.85</td>
<td>1.6</td>
<td>4.004</td>
<td>80.08</td>
<td>7.70979</td>
</tr>
<tr>
<td></td>
<td>BB</td>
<td>52.04</td>
<td>5.204</td>
<td>10.17</td>
<td>3.3</td>
<td>4.0</td>
<td>80.0</td>
<td>13.01</td>
</tr>
<tr>
<td></td>
<td>BBE</td>
<td>67.48</td>
<td>6.748</td>
<td>13.17</td>
<td>4.27</td>
<td>4.92</td>
<td>98.4</td>
<td>13.71545</td>
</tr>
<tr>
<td></td>
<td>DUP</td>
<td>116.25</td>
<td>11.625</td>
<td>19.26</td>
<td>6.25</td>
<td>5.0</td>
<td>100</td>
<td>23.25</td>
</tr>
<tr>
<td>*All</td>
<td></td>
<td>119.24</td>
<td>11.924</td>
<td>20.617</td>
<td>6.68</td>
<td>5.0</td>
<td>100</td>
<td>23.848</td>
</tr>
<tr>
<td>IF-NIP</td>
<td></td>
<td>86.23</td>
<td>8.623</td>
<td>15.74</td>
<td>5.11</td>
<td>4.0</td>
<td>80.0</td>
<td>21.5575</td>
</tr>
</tbody>
</table>

Table 6 – Results for conducting basic coverage maximization.
*Tests selected to cover all of the MC, MCP, BB, BBE and DUP profiles

result of averaging 1000 different executions of the greedy selection algorithm; this explains why the columns showing the number of selected tests and the number of covered defects might contain fractional numbers.

Figures 28, 29 and 30 compare the defect detection performance of basic coverage maximization using various profile types to random sampling. For javac, Figure 28 shows that for all profile types basic coverage maximization performed better than random sampling and as we go up the granularity scale, more tests are selected and more defects are covered. Also, IF-NIP covers more defects with fewer tests selected than the
combined profiles of MC, MCP, BB, BBE and DUP. This means that IF-NIP performed better than when combining data flow (DUP) and control flow (BBE) profiles. For Xerces, Figure 29 shows that basic coverage maximization performed better than random sampling only for Slice-IP and DUP. One possible reason for this unexpected result is the fact that one of the defects in Xerces causes 55 failures, and random sampling is likely to pick up one of those failures with a relatively small sample. Slice-IP did best since it covered the most defects (74.86%) and its ratio of selected tests over covered defects was the lowest (77.52). The other seemingly unexpected result for Xerces was that BBE and DUP covered more defects than IF-NIP. The following example clarifies how BBE and DUP might contain information missed by IF-NIP:

    S1: x = 1;
    S2: if (x > 1) {
    S3:      y = x;
    }
    S4: z = 1;

Following the execution of the snippet of code above, the IF-NIP profiles will not contain any information relevant to it since no flow occurs from any object to another, whereas the DUP profiles will contain information representing the use of x in S2 and the BBE profiles will contain information representing the control flow from S2 to S4. Finally, for JTidy, Figure 30 shows that for all profile types basic coverage maximization performed better than random sampling. Similarly to Xerces, BBE and DUP covered more defects than IF-NIP.
Figure 28 - Defect detection performance of basic coverage maximization and random sampling for *Javac*
Figure 29 - Defect detection performance of basic coverage maximization and random sampling for *Xerces*
Figure 30 - Defect detection performance of basic coverage maximization and random sampling for JTidy
9.5 OBT results and discussions

The OBT experiments used the *proportional binary dissimilarity metric* [15][16], which compares two profiles based on the number of times a profile feature was exercised and on whether that feature was exercised at all. Clustering was performed using the *hierarchical agglomerative* clustering method, which initially places each test in a separate cluster then merges each pair of clusters that is minimally dissimilar into a larger cluster. The process continues until \( k \) clusters remain, for a chosen \( k \). The value of \( k \) was varied to correspond to different percentages of the size of the test suite. The sizes used were 1\%, 2.5\%, 5\%, 10\%, 25\% and 30\% of the test suite. For example, in the case of *Xerces* with 1663 tests, \( k \) was chosen to be 16, 41, 83, 166, 333, and 500.

In the case of OPC sampling, for every program and profile type combination, the experiments involved choosing \( k \), clustering, randomly selecting a single test from each of the \( k \) clusters, and recording the number of failures and defects covered. The process of selecting the tests was replicated 1000 times and the results averages were used, which explains the fractional numbers in the results.

Figures 31, 32, 33, 34, 35, and 36 compare the defect detection performance for one-per-cluster sampling and failure pursuit for *javac*, *Xerces* and *JTidy*. For *javac* and *JTidy*, *IF-NIP* does relatively worse than other profile types for small subset sizes but equally good or better for larger subsets. For *Xerces*, the results for the different profile types are very close to each other for any given subset size. These results are in agreement with the findings presented in [47], i.e., profile types have a relatively small effect on the results for cluster filtering using one-per-cluster sampling or failure pursuit.
Figure 31 - Defect detection performance for one-per-cluster sampling and random sampling for javac
Figure 32 - Defect detection performance for one-per-cluster sampling and random sampling for Xerces
Figure 33 - Defect detection performance for one-per-cluster sampling and random sampling for JTidy
Figure 34 - Defect detection performance for failure pursuit and random sampling for \textit{javac}
Figure 35 - Defect detection performance for failure pursuit and random sampling for Xerces
Figure 36 - Defect detection performance for failure pursuit and random sampling for JTidy
Chapter 10

Strength and length of information flows

In this chapter we will try to answer the following question: Is the length of an information flow indicative of its strength?

We believe that the measure of the statistical dependence of a sink object on a source object in an information flow is a plausible metric for the strength of that flow. Being statistical in nature, such metric cannot be computed on the fly by our tool, but the length of the flow can. The length of an information flow is the length of the chain of the dynamic data and control dependences it comprises. In this chapter we will present methods to statistically measure the strength of a flow. Then we will describe how to compute the length of a flow. Finally, we will empirically investigate the relationship between the strength and length of a flow. If our results reveal a strong (negative) correlation between strength and length, then a user of our tool can judge the strength of a detected illegal flow by its measured length.

10.1 Strength

We will use the computed correlation coefficient between the source and target of a flow as a measure of the statistical dependence of the target on the source and consequently as the strength of the flow. Since the widely used product moment correlation coefficient [41], also known as Pearson’s r or simply the standard correlation coefficient is limited to continuous variables that are linearly related, it is clearly not
always appropriate to use it as is for our purposes. We need to explore other bivariate correlation coefficients [41] and/or ways to overcome the limitations of the standard correlation coefficient.

The correlation ratio or eta coefficient relates a response variable to a categorical predictor, it is therefore suitable to use when the source is binary and the target is of any type (i.e., integer, real or binary). Also, it is known that transforming one or both variables does not impact the relationship between the two variables. Therefore, it is appropriate to use the standard correlation coefficient following a transformation that would make the relationship of two non-linearly related variables as linear as possible.

Note that in the case of complex objects, an integer representation of those objects can be substituted in the computation of the correlation coefficients. For example, the value returned by the Java `hashCode()` method can be used to represent a Java `String` object.

10.2 Length

Since a given source and target object pair can be involved in flows occurring at different locations in a program, we will identify a flow instance by the action that defined its source and the action that defined its target. Let $o_{source}$ be the source and $o_{target}$ the target of a flow instance such that action $s^k$ last defined $o_{source}$ and action $t^m$ last defined $o_{target}$ at the sink of the flow. The length of such flow instance is based on the following equation:
\[
\text{LengthFlow}(s^k, t^m) = \begin{cases} 
1, & o_{\text{source}} \in U(t^m) \\
\text{Min}(\text{LengthFlow}(s^k, r^n)) + 1, & o_{\text{source}} \notin U(t^m) 
\end{cases}
\] (III)

where \( r^n \in \text{DiInfluence}(t^m) \) and \( o_{\text{source}} \in \text{InfoFlow}(r^n) \)

The length is 1 if \( t^m \) uses \( o_{\text{source}} \), otherwise it is the length of the shortest dependence chain through which \( o_{\text{source}} \) flows into \( o_{\text{target}} \).

We compute the length of an information flow primarily by applying equation (III). The inductive nature of equation (III) makes our algorithm forward computing and therefore suitable for online deployment.

### 10.3 Strength vs. length

Learning the length of a detected illegal flow can be valuable only if we first establish that there is a correlation between the strength and length of a flow. For that purpose, we conducted experiments using Xerces, an XML parser, JTidy, an XML pretty printer and a file Diff tool. For each flow instance we stored the quintuplet \((o_{\text{source}}, o_{\text{target}}, l, \text{value}_{\text{source}}, \text{value}_{\text{target}})\) where \( o_{\text{source}} \) is the source identifier, \( o_{\text{target}} \) the target identifier, \( l \) the length of the flow, \( \text{value}_{\text{source}} \) the source value and \( \text{value}_{\text{target}} \) the target value. We then computed the bivariate correlation coefficients for the entries sharing the same \((o_{\text{source}}, o_{\text{target}}, l)\). Finally, we computed the average of the correlation coefficients that were computed at a given length \( l \).

When the source of the flow is binary, we used the \( \eta \) coefficient [19] to compute the correlation between the source and target. The \( \eta \) coefficient, which is free of the assumption of linearity is defined below:
\[ \eta^2 = \frac{\sigma^2_y}{y^2}, \text{ where } y \text{ is the value of the target} \]

When the source of the flow is not binary, we computed a set of standard correlation coefficients each following a given data transformation, then used the maximum of the set as our correlation coefficient. The maximum coefficient corresponds to the transformation that straightened out the curve the most. One transformation we used is the log-log transformation which transforms the target variable \( y \) to \( \log(y) \) and the source variable \( x \) to \( \log(x) \). Below is the list of the other transformations we used; it is a subset of the so-called ladder of powers:

\[ x^{-2}, x^{-1}, x^{-1/2}, \log(x), x^{1/2}, x^1, x^2 \]

Note that all of the above transformations are monotonic and that transformation \( x^1 \) is a no-op. Clearly, many relationships between sources and targets cannot be straightened out noticeably using any of the above transformations; therefore by using the standard correlation coefficient we expect to get lower correlation numbers on average. Such fact is acceptable for our purposes since we are only interested in finding the general shape of the relationship between the length and strength of a flow and not necessarily the exact strength for a given length.

Figures 37 and 38 show the relationship between the length of flows and the average correlation coefficients based on two different Xerces executions. Figure 38 involves an input XML files that is relatively larger and use more complicated features; this fact is reflected in more data points corresponding to higher length values. Figure 39 shows the relationship between the length of flows and the average correlation coefficients for two JTidy executions and one Diff execution. The Xerces data support
the hypothesis that the strength of an information flow is inversely proportional to its length, the JTidy and Diff data sets do not. But all data sets agree on that flows that are short in length are likely to be strong.

In summary, our results show that information flows that are short in length are likely to be strong and flows that are long are not necessarily weak.
Figure 37 – Top: length vs. average correlation coefficient. Middle: length vs. percentage of average correlation coefficient that are zero. Bottom: length vs. percentage of average correlation coefficient that are greater than 0.5
**Figure 38** – length vs. strength data corresponding to a *Xerces* execution on an input file that is larger and more complicated than the one used for Figure 37
Figure 39 – length vs. strength data corresponding to two different JTidy executions (top) and one Diff execution (bottom)
An Intrusion Detection System or IDS is classified according to the infrastructure component it monitors [7], as network-based, host-based or application-based. Our proposed IDS is application-based. It monitors the behavior of an application as it interacts with a particular user or responds to a particular request. We assume that any attack is solely a result of a malicious use of the application’s published API, i.e., the application is running in a trustworthy environment that does not allow for buffer overflows or any other code injection schemes.

An Intrusion Detection System can be further classified as signature-based or anomaly-based. A signature-based IDS, whose goal is *misuse detection*, watches the behavior of its subject for matches with patterns of events (signatures) specific to known attacks. An anomaly-based IDS, whose goal is *anomaly detection*, watches the behavior of its subject for abnormal events (anomalies) that may indicate an attack. It is based on the principle that an attack behavior differs considerably from a normal behavior and that it can be detected by contrasting it to a database of normal behaviors. A clear advantage of anomaly detection over misuse detection is that it is capable of detecting new previously unobserved attacks. Misuse detection is more likely to generate less false positives, though. In this chapter we will discuss how our proposed application-based IDS can be used with either detection scheme. Specifically, we will discuss how using it as a signature-based IDS enables the online detection and prevention of attacks involving
illegal information flows, and using it as an anomaly-based IDS, in conjunction with observation-based testing, enables the offline detection of attacks of various kinds. Note that from now on, an application-based IDS will be referred to as ABIDS.

11.1 Misuse detection

Current signature-based ABIDS systems typically monitor a specific application’s audit logs for known malicious patterns. They are generally easy to evade by varying the attack slightly and consequently affecting the logged events. An important and sizable class of attacks involves illegal information flows. Naturally, our information flow analysis tool can be used as a signature-based ABIDS for detecting and preventing attacks involving illegal flows. In such a setup, the information flow policy object would be responsible for signature matching. A signature in this case comprises the description of the source and target of a single information flow, or the sources and targets of a set or sequence of information flows.

When detecting attacks involving illegal information leaks, our tool has a clear advantage over conventional signature-based intrusion detection systems. First, it monitors the information flows directly as opposed to monitoring lower level events, such as system calls, for symptoms of the attack. Second, it does not rely on a sizable database of signatures; instead it validates observed flow patterns against a supplied information flow policy. Third, it is immune to evasions that are based on padding a known attack’s signature with semantic no-ops [74][78], e.g. given a known illegal flow from object $x$ to object $y$, denoted $x \rightarrow y$, inserting no-op flows involving objects $v$ and $w$
such as to exhibit \( x \rightarrow v \rightarrow w \rightarrow y \), would still be detected by our tool due to information flow transitivity.

In the case when an inserted no-op information flow involves an external, i.e., uninstrumented entity such as the file system, the monitored chain of flows would break and our tool could be evaded. A counter measure to this problem would be to treat with suspicion every flow of secure data to an external target. For example, our tool can be used as a signature-based ABIDS to detect and prevent the attacks on the DefaultServlet application as described in Section 7.4. An appropriate information flow policy would have \( \text{Sources} = \text{SensitiveSources} = \{ \text{objects of type java.io.File or java.io.FileInputStream associated with files not belonging to public directories} \} \) and \( \text{Sinks} = \{ \text{write and print methods of PrintWriter and ServletOutputStream, write methods of java.io.OutputStream} \} \). This policy would detect and prevent any flow of data from a sensitive input file to the client or to any output stream. Note that attempts to evade our tool by padding no-op flows to external entities would most likely be detected and prevented as a result of making the \text{write} methods of \text{java.io.OutputStream} members of \text{Sinks}.

In summary, signature-based IDS systems are generally easy to evade using polymorphic attacks, i.e., attacks that vary slightly from one instance to another, but our system is immune to such evasions (given that they involve illegal flows) since it monitors only the events that are essential to the attack, i.e., source reads and target writes.

11.2 Anomaly detection
In many cases it is not realistic to assume that all possible attacks along with their manifested behaviors are known; anomaly-based IDS systems are most suitable in such cases. Typically, detecting anomalies involves online detection of deviations from a model of normal behavior followed by generation of alarms. One way to evade such detection scheme is to somehow incorporate the intended malicious behavior into the training data of the model of normal behavior, i.e., modify what is considered to be normal. Another way is to alter the malicious behavior so that it would go undetected by the IDS, i.e., modify the attack to make it look like normal.

Our proposed approach to anomaly detection is significantly different, as it does not rely on building a model of normal behavior or self [23]. It is based on the premise that observation-based testing through cluster analysis is capable of detecting attack anomalies. Dickinson et al have already shown experimentally that failures often have unusual profiles that are revealed by cluster analysis and that failures cluster together and often form small clusters in sparsely populated areas of the profile space [15][16]. We believe that attacks would exhibit similar traits in the profile space.

Since observation-based testing requires a large number of execution profiles generated online or by replaying captured executions, our proposed approach would detect anomalies offline, i.e., after the fact. Unlike our proposed signature-based ABIDS, which is only capable of detecting attacks involving illegal flows, our proposed anomaly-based ABIDS is capable of detecting various types of attacks since it watches for any suspicious behavior and not just illegal flows. Note though that a particular type of attack might be more likely to be detected using a specific profiling scheme. For example,
Illegal flow attacks are more likely to be detected using information flow profiling than any other type of profiling.

In order to experimentally demonstrate that security attacks exhibit unusual profiles, we devised a test suite for the Java application server Apache Tomcat [76] that includes a known security attack. The replicated source disclosure attack is described and tagged with id number 5580 in the Open Source Vulnerability Database [57]. It involves a remote attacker sending a GET request that does not end with an HTTP protocol specification (HTTP/1.0 or HTTP/1.1), which will disclose the source code of the requested JSP file as opposed to the output of its execution. For example, invoking the date.jsp using a) returns the normal output of the JSP, whereas invoking it using b) causes the source code to be returned instead:

a) "GET /examples/jsp/dates/date.jsp HTTP/1.1"

b) "GET /examples/jsp/dates/date.jsp"

We instrumented the Tomcat application server and submitted 150 requests to it that included:

1) 20 requests replicating the attack
2) 80 requests involving the same JSPs as in 1) except that those requests ended with an HTTP protocol, i.e., normal requests
3) 50 other requests involving various JSPs and servlets

Figure 40 shows the multidimensional scaling plot (MDS) [48] of the dissimilarity metrics for the 150 execution profiles. MDS is a technique that produces a plot where the distance between points approximate the distance given by the dissimilarity metric. The top rounded rectangle encloses the execution profiles of the 20 attacks and the larger
rounded rectangle encloses the execution profiles of the corresponding 80 normal requests. The remaining data points represent the remaining 50 requests. Based on the MDS plot, the attacks executions exhibited profiles that clearly distinguished them from the rest of the executions. In the future, we intend to use more subject programs and more attacks to evaluate the effectiveness of information flow and slice profiling in conjunction with OBT for anomaly detection.
Figure 40 - MDS plot of the Tomcat executions based on information flow profiles
Chapter 12

Future work

Currently our algorithms do not address intra-procedural and inter-procedural control dependences resulting from `halt/exit` instructions [69] or from exceptions. We plan to augment our algorithms and tool to provide this capability, e.g., by modifying the `DDynCD` algorithm to employ an extended form of control flow graph similar to that used for static inter-procedural control dependence analysis in [69]. Next, we plan to address the limitations of our implicit flow detection algorithm as described in Chapter 8 and conduct experiments with the implicit flow detection option enabled. More subject programs need to be used to further evaluate the effectiveness of information flow and slice profiling in test case filtering and anomaly detection. We intend to investigate the application of our approach in transactional systems and in the absence of an information flow policy.

We plan to investigate ways to improve the performance of our tool so that online deployment would be more widely feasible. We intend to enhance our tool by providing a more convenient interface for configuration of information flow policies and by allowing runtime configuration of policies, which would facilitate monitoring of long-running programs. We also plan to integrate our tool with popular IDE’s and debuggers such as `Eclipse` and investigate a .NET implementation. Finally, we plan to address the limitations of the capture/replay tool `jRapture`, as described in Section 2.4.
Chapter 13

Conclusions

We presented new algorithms for dynamic information flow analysis and dynamic slicing. They are the first precise forward-computing algorithms for these problems to be proposed that apply to both structured and unstructured programs. The fact that they are forward computing allows them to be used online when performance is not critical and enables interactive debugging of unsafe information flows and other failures. One unique and optional feature of our algorithms enables them to detect most implicit influences.

We have described an implementation of our algorithms that works with Java byte code programs. It is apparently the first information flow analysis tool to support configurable information flow policies. The functionality and performance of the tool were characterized by employing it in several cases studies. The performance results confirmed that the tool is suitable for online use with applications that are not computationally intensive.

We empirically evaluated the effectiveness of information flow and slice profiling in several test case filtering schemes, namely basic coverage maximization, cluster filtering using one-per-cluster sampling and failure pursuit.

We empirically showed that information flows that are short in length are likely to be strong and flows that are long are not necessarily weak.

We showed how our tool could be used as a signature-based IDS for the online detection and prevention of attacks involving illegal information flows. We also
discussed how it could be used in conjunction with observation-based testing as an anomaly-based IDS for the offline detection of attacks of various kinds.
Chapter 14

Appendix A

Here we present the role of the Instrumenter, then present the main data structures in the Profiler and finally describe the implementation of the main methods in the Profiler.

14.1 Instrumenter

Using BCEL [8] the Instrumenter inserts a number of method invocations to the Profiler at some given points of interest in the subject program. At runtime, the instrumented application invokes the Profiler passing it the necessary information that enables it to monitor information flows and build program slices. The following enumerates those methods along with their respective insertion location. (Note that Figure 41 classifies the various Java byte code instructions into types of statements that we reference below):

a. handleStore: inserted prior to store statements (StoreStatement) of object fields, static fields and array elements.

b. handleLoad: inserted prior to load statements (LoadStatement) of object fields, static fields and array elements.

c. handleLocalStore: inserted prior to store statements (StoreStatement) of local variables.
d. handleLocalLoad: inserted prior to load statements of local variables (LocalLoadStatement).

e. handlePredicate: inserted prior to a predicate statement (IfSelectStatement).

f. handleMisc: inserted prior to a statement of type MiscStatement that is a consumer and/or a producer (a statement that pops one or more values off the JVM operand stack is called a consumer. A statement that pushes one or more values on the JVM operand stack is called a producer).

g. handlePreMethodInvoke: inserted prior to a method invocation (InvokeStatement).
h. **handleMethodEntry**: inserted at the beginning of a method.

i. **handleMethodExit**: inserted prior to a method exit, i.e., prior to a return statement (**ReturnStatement**).

j. **handlePostMethodInvoke**: inserted following a method invocation.

k. **handleClassLoad**: inserted at the entry point of the static block of a Java class. Note that **handleClassLoad** will be called only once, at class load time.

In addition, when the implicit flow detection option is enabled, the **Instrumenter** appropriately inserts **addFlow** calls to the **Profiler** and in some cases modifies the control flow graph of a given method by adding new vertices, as described in Chapter 8.

### 14.2 **Profiler: Data Structures**

The following are the primary data structures used in the **profiler’s** implementation:

1) **DataBag** (see Figure 42): a **DataBag** has a one-to-one relation with an action, it stores the **DInfluence** set associated with that action and primarily computes and stores the **InfoFlow** and **DynSlice** sets for that action.

2) **MethodDesignator** (see Figure 43): identifies a given method based on its class name, method name and signature.

3) **VarDesignator** (see Figure 43): identifies a given variable $v$. For example, if $v$ is a static field, this object stores the class name and the field name; if $v$ is a
local variable, it stores an instance of MethodDesignator, the thread identifier and the variable index.

4) For a given method $m$, the following entities are created locally within a thread:
   a. $CDSTACK(m)$ is a stack used to compute the $DDynCD$ relationships as described in Section 4.1. Instead of stacking decision actions, it stacks the IfSelectDataBag instances that are related to them.
   b. $OPSTACK(m)$ is a stack that loosely parallels the JVM operand stack. Instead of stacking variables (operands) it stacks instances of DataBag. $OPSTACK(m)$ is used to dynamically compute the direct data dependences ($DDynDD$).
   c. $LVARRAY(m)$ is an array where the leading elements (of type DataBag) correspond to the passed parameters of a method (see Definition 13); the remaining elements, of type VarDataBag, correspond to the local variables declared inside the method.

5) $LVSTACK$ is a stack created locally within a thread such that on method entry the algorithm pushes an instance of $LVARRAY$ on it then pops it on method exit. Therefore, the top of $LVSTACK$ stores the DataBag instances of the local variables in the active stack frame.

6) $RETHASH$ is a hash table created locally within a thread; it stores instances of ReturnDataBag that are associated with return actions. An instance of ReturnDataBag is stored in $RETHASH$ on the called method exit, then removed and used by the caller method (see Definition 14).
7) \textit{GLOBHASH} is a hash table shared by all threads; it maintains \texttt{VarDataBag} objects associated with object fields, static fields or array elements (i.e., non local variables). It is the channel through which data flows between threads.

Note that at any point during execution, \textit{GLOBHASH} stores \texttt{InfoFlow} and \texttt{DynSlice} sets for all global symbols, \textit{LVSTACK} stores \texttt{InfoFlow} and \texttt{DynSlice} sets for all local symbols, and the \textit{LVARRAY} (on top of \textit{LVSTACK}) stores \texttt{InfoFlow} and \texttt{DynSlice} sets for the active local symbols. Examining the content of \textit{LVARRAY}, \textit{LVSTACK} and \textit{GLOBHASH} is clearly very valuable when debugging insecure information flows.

14.3 \textbf{Profiler: Methods’ Implementations}

We now provide a brief description of the main methods of the \textit{profiler}. Most of the described methods are to be directly invoked by the instrumented program; others such as \texttt{ComputeInfoFlowAndDynSlice}, \texttt{AddDDynCD}, \texttt{AddDDynDD} and \texttt{AddInterprocCD} are local to the profiler. In our implementation we mostly deal with \texttt{DataBag}s, which have a one-to-one relation with actions. Therefore, when referring to a \texttt{DataBag} instance, the action associated with it is implied.

1) \texttt{AddDDynCD(DataBag db)}

- Retrieves \texttt{CDSTACK(m)} from the thread local area (since each thread will have its own instance of \texttt{CDSTACK(m)}).
- Executes the algorithm of Section 4.1. The returned \texttt{db’} is a decision \texttt{DataBag} such that \texttt{db DDynCD db’}.
- Adds \texttt{db’} to \texttt{db.DInfluence}.
2) AddDDynDD(DataBag db)
   - Retrieves $OPSTACK(m)$ from the thread local area (since each thread will have its own instance of $OPSTACK(m)$).
   - Manages (with a series of pops and pushes) $OPSTACK(m)$ according to the production and consumption of $db$. The consumed (or popped off) DataBags are used by $db$.
   - Adds the consumed DataBags to $db.DInfluence$ ($db$ is $DDynDD$ or $ParamD$ on them).

3) AddInterprocCD(DataBag db)
   - Adds the caller’s DataBag to $db.DInfluence$ ($db$ is $InterprocCD$ on it).

4) ComputeInfoFlowAndDynSlice(DataBag db)
   - Executes the algorithm in Section 5.1. Note that at the time when this method gets called, $db.DInfluence$ already contains all the DataBags that directly influence $db$.

5) handleClassLoad(String className)
   - This method is called when a class is loaded; this sets the stage for the Profiler to build the control flow graphs and immediate postdominance relationships of its methods. Note that such computation is performed the
first time a given method is invoked, i.e., no computation takes place for methods that do not get invoked.

6) handleStore(VarDesignator d)
   - Retrieves db from GLOBHASH; db is an instance of VarDataBag that corresponds to d. If not found, i.e., d is being defined for the first time, creates a new VarDataBag and store it in GLOBHASH.
   - Calls AddDDynCD(db).
   - Calls AddDDynDD(db).
   - Calls AddInterprocCD(db).
   - Calls ComputeInfoFlowAndDynSlice(db).

7) handleLoad(VarDesignator d)
   - Retrieves db from GLOBHASH; db is an instance of VarDataBag that corresponds to d. handleStore guarantees that db will be found.
   - Calls AddDDynCD(db).
   - Calls AddDDynDD(db).
   - Calls AddInterprocCD(db).

8) handleLocalStore(VarDesignator d)
   - Retrieves db from LVARRAY (top of LVSTACK); db is an instance of VarDataBag that corresponds to d. If not found, i.e., d is being defined for the first time, creates a new VarDataBag and store it in LVARRAY.
- Calls AddDDynCD(db).
- Calls AddDDynDD(db).
- Calls AddInterprocCD(db).
- Calls ComputeInfoFlowAndDynSlice(db).

9) handleLocalLoad(VarDesignator d)
- Retrieves db from LVARRAY (top of LVSTACK); db is an instance of VarDataBag that corresponds to d. handleLocalStore guarantees that db will be found.
- Calls AddDDynCD(db).
- Calls AddDDynDD(db).
- Calls AddInterprocCD(db).

10) handlePredicate(IfSelectStatement s)
- Creates db, an instance of IfSelectDataBag, and associates it with s.
- Calls AddDDynCD(db).
- Calls AddDDynDD(db).
- Calls AddInterprocCD(db).
- Calls ComputeInfoFlowAndDynSlice(db).

11) handleMisc(MiscStatement s)
- Creates db, an instance of MiscDataBag, and associates it with s.
- Calls AddDDynCD(db).
- Calls AddDDynDD(db).
- Calls AddInterprocCD(db).
- Calls ComputeInfoFlowAndDynSlice(db).

12) handlePreMethodInvoke (InvokeStatement s)

- Creates db, an instance of InvokeStatement, and associates it with s.
- Calls AddDDynCD(db).
- Calls AddDDynDD(db).
- Calls AddInterprocCD(db).
- Creates an instance of LVARRAY and orderly initializes its leading elements with the DataBag’s corresponding to the passed parameters (popped off OPSTACK) so that the called context will have access to them later in handleMethodEntry.

13) handleMethodEntry()

- Pushes LVARRAY that was created in handlePreMethodInvoke onto LVSTACK.

14) handleMethodExit()

- Creates db, an instance of ReturnDataBag. db will carry the InfoFlow and DynSlice sets associated with the return statement.
- Calls AddDDynCD(db).
- Calls AddDDynDD(db).
• Calls AddInterprocCD(db).
• Calls ComputeInfoFlowAndDynSlice(db).
• Stores db in RETHASH to be used in handlePostMethodInvoke.
• Pops LVARRAY off LVSTACK.

15) handlePostMethodInvoke(MethodDesignator designatorOfCalledMethod)

• Using designatorOfCalledMethod retrieves db’ from RETHASH; db’ is the ReturnDataBag previously stored by handleMethodExit.
• Creates a new db, an instance of InvokeDataBag, adds db’ to db.DInfluence and calls ComputeInfoFlowAndDynSlice(db).
Figure 42 – Class diagram classifying the various DataBags. There is a one-to-one association between a DataBag and an action, for example, an InvokeDataBag is associated with an InvokeStatement action.
Figure 43 – Class diagram classifying the various variable designators (identifiers). In addition, a MethodDesignator identifies a method, a MethodThreadPair identifies a method across thread instances.
Chapter 15

Appendix B

We first define direct dynamic implicit dependence:

**Definition:** Let \( s^k \) and \( t^m \) be two actions in an execution trace \( T \), where \( k < m \), \( s^k \) is a predicate action and \( ipd(s) \) dominates \( t \). Let \( S \) be the set of variables that are used by \( t^m \) and are defined in the static scope of \( s \). Then \( t^m \) is directly dynamically implicitly dependent on \( s^k \), denoted \( t^m DDynImplicitD s^k \), iff \( \exists \) at least one variable in \( S \) that is not defined by \( T(k+1, m-1) \), i.e.:

\[
(D(StaticScope(s)) \cap U(t^m)) - D(k + 1, m - 1) \neq \emptyset
\]

The direct influence relation due to explicit and implicit dependences becomes:

\[
DInfluence'(t^m) = DInfluence(t^m) \cup DDynImplicitD(t^m)
\]

OR

\[
DInfluence'(t^m) = DDynDD(t^m) \cup DDynCD(t^m) \cup ReturnD(t^m) \cup ParamD(t^m) \cup InterprocCD(t^m) \cup DDynImplicitD(t^m)
\]

The **Influence** and **InfoFlow** equations due to explicit and implicit dependences become:

\[
Influence'(t^m) = DInfluence'(t^m) \cup \bigcup_{s^k \in DInfluence(t^m)} Influence'(s^k)
\]

\[
InfoFlow'(t^m) = U(t^m) \cup U(Influence'(t^m))
\]

\[
InfoFlow'(t^m) = U(t^m) \cup \bigcup_{s^k \in DInfluence(t^m)} InfoFlow'(s^k)
\]
In order to prove that our transformation algorithm enables our DIFA algorithm to detect all implicit information flows in addition to explicit ones, we simply need to show that given $s^k$ and $t^m$ such that $t^m \text{DDynImplicitD} s^k$, our transformation algorithm registers in our DIFA algorithm the direct influence of $s^k$ on $t^m$:

Let $s^k$ and $t^m$ be two actions in an execution trace $T$ such that $t^m \text{DDynImplicitD} s^k$.

Therefore the following hold:

1) $s$ is a predicate

2) $ipd(s)$ dominates $t$

3) $\exists$ a variable $x$ in $S$ that is not defined by $T(k+1, m-1)$, where $S$ is the set of variables that are used by $t^m$ and are defined in the static scope of $s$.

Our transformation algorithm inserts at the start of each branch of $s$ addFlow calls reflecting the explicit influences due to $s$ that the execution might have encountered if the alternative branches were taken. $x$ is not defined in the branch taken by $T(k+1, m-1)$ then it must be defined in an alternative branch; therefore, our transformation algorithm inserts addFlow($s$, x_id) in the branch taken by $T(k+1, m-1)$. The execution of addFlow($s$, x_id) in $T(k+1, m-1)$ registers the influence of $s^k$ on $x$ and consequently on $t^m$ since $t^m$ uses $x$.

Note: If $s^k$ belongs to $\{Influence'(t^m) - Influence(t^m)\}$ then the influence of $s^k$ on $t^m$ is purely implicit. Similarly, if $x$ belongs to $\{InfoFlow'(t^m) - InfoFlow(t^m)\}$ then the flow from $x$ to $t^m$ is purely implicit. This basically means that if DDynImplicitD was omitted from DInfluence' then the dynamic chain of dependence between $s^k$ and $t^m$ will be broken.
References


[58] Palsberg J. and Schwartzbach M. Object-Oriented Type Inference. ACM OOPSLA (October 1991).

[59] Plevyak J. and Chien A. Precise Concrete Type Inference for Object-Oriented Languages. ACM OOPSLA (October 1994).


[74] Tan K., Killourhy K., and Maxion R. Undermining an anomaly-based intrusion detection system using common exploits. 5th International Symposium on Recent Advances in Intrusion Detection (Zurich, October 2002).


[77] Transforming variables: http://www.statmaster.sdu.dk/courses/st111/Module04


[88] www.securityfocus.com/bid/5786/info