DOCTORAL THESIS

Experiments with property driven monitoring of C programs

Vorobyov, Kostyantyn

Award date:
2015

DOCTORAL THESIS

Experiments with property driven monitoring of C programs

Vorobyov, Kostyantyn

Award date:
2015

Awarding institution:
Bond University

Experiments with Property Driven Monitoring of C Programs

Presented by

Kostyantyn Vorobyov

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

Department of Informatics
Bond University
Australia
July 2015

Typeset in LaTeX 2$_\varepsilon$.

Abstract

Monitoring is a dynamic technique that observes executions of programs and detects errors at runtime. The key issue in monitoring is performance overhead: if the overhead is too high, then monitoring takes too long to terminate or runs out of system resources.

This thesis investigates aspects of monitoring for bug detection, seeking implementations that identify errors precisely and that have overheads acceptable for use with testing. These concerns are addressed by instantiating monitoring analyses for the problems of memory leaks and disclosure of confidential information, and using experimentation to investigate incurred overheads. A generalisation is then provided over primitives identified during the instantiations. This enables monitoring via abstract specifications aimed at reducing the cost of developing monitoring analysis from scratch.

First, this thesis presents an approach to memory leak detection. The key issue addressed by the author's technique is detection of locations where the leaked memory was lost. Leak detection is enabled by tracking each allocated memory block and computing the dereference of the block's address space. This represents reachability of the memory block with respect to program variables. Unreachable blocks that have not been de-allocated by the program are reported as memory leaks at the end of execution. The locations of leakage are given by the locations where blocks were last reachable. The results of experimentation with the prototype implementation for C programs indicate that, for monitoring test suites of UNIX utilities and programs selected from the Standard Performance Evaluation Corporation (SPEC) CPU datasets, the overheads of the present approach compare favourably to the results produced by the Valgrind memory debugger.

This thesis also describes an approach to preventing leakage of sensitive information used by a program. The technique analyses values and has the ability to identify whether a disclosed value represents an information leak with respect to the values considered secret at runtime. This is enabled by tracking secret values and checking whether assignments transfer secret values to publicly observable locations. A prototype implementation for C programs was used to analyse security-oriented UNIX utilities and programs chosen from the SPEC CPU datasets. The results of the experiments indicate that the overhead required to detect password disclosure in real

software does not exceed 1%. The overheads associated with detection of Common Weakness Enumeration security vulnerabilities in real applications and SPEC CPU programs are higher, but remain acceptable for use with testing.

Finally, this thesis presents a mechanism, called Specification for Monitoring (SFM), for concise specification of monitoring analysis. The strength of SFM is that it separates semantic issues related to monitoring from the implementation details. This separation of concerns results in compact specifications, as the implementation details are delegated to the implementation of API. Further, although SFM is abstract, it is less so than many specification techniques, which means that implementation of the API can still be very efficient. Several specifications are presented to illustrate the key ideas.

Declaration

This thesis is submitted to Bond University in fulfilment of the requirements of the degree of *Doctor of Philosophy*. This thesis represents my own original work towards this research degree and contains no material which has been previously submitted for a degree or diploma at this University or any other institution, except where due acknowledgement is made.

Kostyantyn Vorobyov
Date: July 1, 2015

Department of Informatics
Bond University
Robina 4229
Australia

Acknowledgements

First, I would like to express my deepest gratitude to my advisor, Dr. Padmanabhan Krishnan, for his mentoring, patience, guidance and support throughout all stages of this project. I am greatly appreciative of his commitment to his students and deep involvement in my PhD even after he left Bond University. I would also like to thank Dr. Krishnan for financially supporting my research via the Centre for Software Assurance fund and for giving me an opportunity to be involved in research projects outside of this PhD.

I would like to thank Dr. Marcus Randall and Dr. Phil Stocks who also supervised my PhD. I am very grateful for their guidance and support.

I am incredibly thankful to Oracle Labs in Brisbane (formerly Sun Microsystems Laboratories) for funding this PhD. Without their financial support, completing this thesis would be an extremely difficult task. I am very grateful to all staff at Oracle Labs and, especially, to the Research Director, Dr. Cristina Cifuentes, for many opportunities to present my research, continuous feedback and countless useful discussions. Finally, I am greatly thankful of the invaluable time spent at Oracle Labs during my two internships there.

Many thanks go to the former School of Information Technology at Bond University for a truly enjoyable time and financial support for conference travel.

I am grateful to Dr. Matt Carter and Vandy Mau for an opportunity to work at MFDC, which allowed me to support myself while finalising this thesis. I would also like to thank my colleagues at the Thought Fort co-working space, where I spent last seven months.

I appreciate editorial advice provided by Elite Editing. This editorial intervention was restricted to Standards D and E of the Australian Standards for Editing Practice.

Last, but not the least, I would like to thank my mother and my sister for their love, unconditional support and their belief in me. I cannot put to words the level of sincere gratitude and appreciation I feel; without them this thesis would not be possible. I am also endlessly grateful to my late father, whose dedication and integrity have been an inspiration to me all these years.

Contents

1 Introduction **1**

1.1 Motivation and Problem Statement 1

1.1.1 Monitoring for Bug Detection 3

1.1.2 Memory Leaks 4

1.1.3 Information Leakage 5

1.2 Aims and Scope 6

1.3 Research Questions 7

1.4 Contributions 7

1.5 Thesis Structure 9

2 Literature Review **11**

2.1 Memory Leaks 11

2.1.1 Memory Debuggers 12

2.1.2 Detecting Locations of Leakage 13

2.1.3 Dynamic Sampling 15

2.1.4 Detecting Memory Leaks at the Hardware Level 16

2.2 Information Leakage 16

2.2.1 Language-based Information Flow Security 17

2.2.2 Data Flow Tracking 17

2.2.3 Information Flow Analysis 18

2.2.4 Dynamic Taint Analysis 19

2.2.5 Secure Executions 20

2.2.6 Testing 20

2.3 Monitoring Specifications 21

2.3.1 Low-Level Instrumentation 21

2.3.2 Monitoring Program Events 22

2.3.3 Behavioural Interface Specification Languages 23

2.3.4 Runtime Verification 25

2.3.5 Trace Monitors 27

3 Preliminaries **30**
3.1 Syntax . . . 30
3.2 Memory Semantics . . . 31
3.3 Operational Semantics . . . 32
3.3.1 Evaluation of Expressions . . . 32
3.3.2 Operational Semantics of Program Commands . . . 32

4 Detection of Memory Leaks and Locations of Leakage **34**
4.1 Syntax and Semantics . . . 35
4.1.1 Syntax . . . 36
4.1.2 Memory Semantics . . . 36
4.1.3 Operational Semantics . . . 37
4.2 Memory Leak Detection . . . 43
4.2.1 Memory Tracking State . . . 43
4.2.2 Semantics of Monitoring Commands . . . 44
4.2.3 Syntactic Transformations . . . 47
4.2.4 Execution of Instrumented Programs . . . 49
4.3 Application on C Programs . . . 50
4.3.1 Memory Blocks . . . 50
4.3.2 Labels . . . 50
4.3.3 Memory Tracking . . . 50
4.3.4 Memory Allocation and De-allocation . . . 51
4.3.5 Memory Leak Reporting . . . 51
4.4 Empirical Evaluation . . . 51
4.4.1 Objectives . . . 52
4.4.2 Experiment Setup . . . 52
4.4.3 Memory Leak Reports . . . 53
4.4.4 Performance Overheads . . . 53
4.4.5 Threats to Validity . . . 61
4.5 Detecting Illegal Memory Modifications . . . 61
4.5.1 Extension at the Abstract Level . . . 62
4.5.2 Application on C Programs . . . 63
4.5.3 Experimentation Results . . . 64
4.6 Concluding Remarks . . . 66

5 A Value Tracking Approach to Information Flow Security **68**
5.1 Syntax and Semantics . . . 70
5.1.1 Syntax . . . 70
5.1.2 Memory Semantics . . . 71
5.1.3 Operational Semantics . . . 71
5.1.4 Information Leak . . . 74

5.2 Information Leak Detection 75
5.2.1 Monitoring State 75
5.2.2 Semantics of Monitoring Commands 75
5.2.3 Transformation Rules 78
5.2.4 Execution of Instrumented Programs 80
5.3 Application to C Programs 80
5.4 Experimental Results 82
5.4.1 Objectives 83
5.4.2 Experimental Setup 84
5.4.3 Password Flow 85
5.4.4 CWE-based Security Properties 86
5.4.5 Threats to Validity 90
5.5 Concluding Remarks 91

6 Concise Specification Language for Monitoring 93
6.1 Need for Generalisation 93
6.2 The SFM Language 96
6.2.1 Informal Model 96
6.2.2 Actions 97
6.2.3 Events 98
6.2.4 Patterns 98
6.3 Monitoring API 100
6.4 SFM Examples 102
6.4.1 Stack Overflow Detection 102
6.4.2 Explicit Information Flow 103
6.4.3 Resource Leakage 104
6.4.4 Detection of SQL Injections 105
6.5 Concluding Remarks 106

7 Summary and Future Work 108
7.1 Summary of Contributions 108
7.1.1 Detection of Memory Leaks and Leakage Locations 108
7.1.2 A Value Tracking Approach to Information Flow Security 110
7.1.3 Common Specification Language 111
7.2 Future Work 112
7.2.1 Improving Overhead Results 112
7.2.2 Using Different Properties 113
7.2.3 Generating Monitoring Analysis 113

A Grammar of the SFM language 114

B Acronyms 116

List of Figures

3.1 Standard Abstract Language 31
3.2 Evaluation of Program Expressions 32
3.3 Operational Semantics of Program Commands 33

4.1 Abstract Language Extended with Dynamic Memory Allocation 36
4.2 Evaluation of Expressions 38
4.3 Operational Semantics of Program Commands 39
4.3 Operational Semantics of Commands (cont.) 40
4.4 Operational Semantics of Monitoring Commands 45
4.5 Syntactic Transformations 48
4.6 `art`: Valgrind Report 53
4.7 `art`: Skiff Report (**Full** mode) 54
4.8 `twolf`: Skiff Report (**Full** mode) 54
4.9 `locate`: Valgrind Report 54
4.10 `locate`: Skiff Report (**Minimal** mode) 54
4.11 Valgrind vs. **Minimal** Mode. Memory Overhead 55
4.12 Valgrind vs. **Minimal** Mode. Runtime Overhead 56
4.13 UNIX Programs Runtime Overhead 57
4.14 UNIX Programs Memory Overhead 58
4.15 SPEC CPU Runtime Overhead 58
4.16 UNIX Programs Overhead Relative to Memory Usage 59
4.17 SPEC CPU Overhead Relative to Memory Usage 59
4.18 SPEC CPU Overhead Relative to Memory Usage 60
4.19 Operational Semantics of a **checkDereference** 63
4.20 Syntactic Transformations for Illegal Dereference Detection 63
4.21 Valgrind vs. Skiff. Runtime Overhead 65

5.1 Abstract Language 70
5.2 Evaluation of Program Expressions 72
5.3 Operational Semantics of Program Commands 73
5.4 Operational Semantics of Monitoring Commands 76
5.5 Transformation Rules 78

5.6 Runtime Overheads of UNIX Utilities 85
5.7 Runtime Overheads of Programs from SPEC CPU Datasets 88
5.8 Runtime Overheads of Security Programs 89

List of Tables

5.1 Instrumentation Statistics of UNIX Utilities 86
5.2 Instrumentation Statistics of Programs from SPEC CPU Datasets 87
5.3 Instrumentation Statistics of Security Programs 89

6.1 Program Events 99
6.2 SFM API 101

1 Introduction

1.1 Motivation and Problem Statement

The effects of software errors (often referred to as bugs) differ in severity. While some are relatively harmless, others often lead to such serious problems as data corruption, breaches of security, disclosure of confidential information, performance deterioration and system crashes. Given that software is widely used (including safety critical systems), detecting computer bugs has long become very important.

Nowadays, software has grown large and complex, often spanning over hundreds of thousands and even millions lines of code. As a result, manual detection of errors in large programs is a non-trivial task. Issues associated with manual error detection are often solved using program analysis, an approach capable of detecting errors with no or little human interaction.

Program analysis is a broad area that focuses on automatic analysis of program behaviours with respect to a given property (or a set of properties). For example, a program can be analysed to ensure that it fulfils certain requirements (e.g., check if a program is compliant with a given language standard). Then, an error formalised as a property (for instance, using a code pattern that represents the error) can be also be detected by means of program analysis.

Program analysis detects software bugs either statically (without executing a program) or dynamically (at runtime). A static approach detects errors in a program by examining the states of its abstract representation, an approximation to the real program. The benefit of using a static technique, is that it can identify errors without user

interaction or code instrumentation. The precision of the analysis, however, depends on the approximations made by the analyser. The complexity of a precise static technique for solving problems such as aliasing, which is common to many programming languages, ranges between NP-hard and undecidable [1]. As a result, precise static analysis of large programs is often not feasible in practice.

Static techniques that use imprecise approximations can check large code bases in reasonable time, but typically report alarms that do not correspond to real errors or miss bugs. False positive or false negative results (i.e., incorrectly reported or missed errors) are produced due to the imprecise approximations used. Therefore, a developer or an analyst, who receives a potentially incomplete bug report, also needs to identify whether the reported results correspond to real errors. Verifying bug reports in large programs is not a trivial task.

The trade-offs of using static techniques of different precision for bug detection are demonstrated in the preliminary work [2]. This work compares model checking and static program analysis for a buffer overflow problem using an empirical evaluation of the tools that implement these techniques – CBMC [3] (a model checker) and Parfait [4] (a static program analysis tool) . Even though the authors used small programs, the runtime and memory usage of CBMC were significantly greater than those of Parfait. For example, to check benchmarks from the Iowa suite [5], CBMC used over 18 hours and 2.4 GB of memory, while Parfait checked the same programs in under one minute with less than 7MB of memory. However, CBMC was 98% accurate, while Parfait missed 35% of the seeded bugs. The present findings indicate that, even though model checking was more accurate, it is unlikely to scale to full-sized systems. A similar study, consistent with these results, was reported by Engler [6].

Author's further experimentation [7] with detection of buffer overflow errors using symbolic execution gave an indication that issues associated with scalability and precision of static methods can be addressed using a dynamic approach. One such dynamic technique is monitoring, which observes executions of a program and detects errors at runtime. Since monitoring operates on instantiated values, problems such as aliasing rarely result in infeasible computations. However, unlike static analysis, which considers all program behaviours, monitoring checks only individual executions.

A key issue in monitoring is the cost of analysis. This cost, also known as performance overhead, accounts for additional resources (e.g., memory or CPU consumption) and time required to complete the analysis. If the overhead is too high, the analysis does not scale, so that monitoring takes too long to terminate or runs out of system resources.

It is not atypical for a monitoring technique to incur runtime overheads of tens of times those of the normal execution [8–13]. This is because a monitoring analysis

often requires tracking large amounts of data and checking many operations performed by the program. For example, the Memcheck tool of the Valgrind memory debugger [14] keeps track of all memory allocated by the program and checks each memory read or write operation with respect to the captured allocation. As a result, the memory consumption of the program under Memcheck supervision doubles. Further, the experimentation with computationally expensive programs [14] indicates that the runtime overheads of Memcheck for individual runs average 26.5 times those of the unobserved execution, and can be as high as 47.8 times. Nevertheless, due to the value of the Memcheck monitoring analysis, which precisely identifies issues in memory safety, such overheads are deemed acceptable for use with testing. This is evidenced by the application of Memcheck to testing large and complex projects such as Mozilla, OpenOffice, MySQL, GIMP and many others [15].

The overheads associated with monitoring can be reduced by using only an approximation to the actual information required. For example, the AddressSanitizer memory debugger [16] assumes that all memory accesses are aligned at an 8-byte boundary. This allows it to shadow the virtual address space of a program using one-eighth of its original size. However, such approximations are prone to missing bugs. Similarly, sampling techniques used to track memory leaks [17–19] assume that the memory regions that are not accessed for a "long time" leak memory. This approach reduces runtime overheads (compared to conventional techniques that capture memory and track accesses and allocations); however results in an analysis that is not sound. Imprecise approximations in monitoring can also lead to false alarms. For example, a technique for detecting locations of memory leaks by Maebe et al. [20] reports both false positive and false negative results. This is because this approach handles the majority but not all of the cases required to detect leaks precisely. Maebe et al. indicate that accurate analysis is possible, but likely to result in greater overheads.

In summary, the overheads of monitored executions vary based on the precision of the approximations used by the monitor. A more precise approximation typically results in greater overheads (where 20-40 times slowdown is still considered acceptable for use with testing), but greater accuracy in identifying issues during a run of a program and vice versa. Imprecise approximations result in cheaper analysis, but raise false alarms or miss bugs.

1.1.1 Monitoring for Bug Detection

Research in the field of precise and scalable monitoring for bug detection often focuses on specific types of defects, such as buffer overflows or SQL injections. This is because detection of different types of errors potentially requires different implementation techniques. Therefore, addressing a small issue is more likely to yield an efficient implementation, as it can be tailored to specific needs.

The following section discusses some of the limitations of monitoring for specific problems. First, it deals with issues related to dynamic detection of memory leaks in languages where memory is not managed. It then then describes limitations associated with discovering disclosure or sensitive information at runtime.

1.1.2 Memory Leaks

A memory leak is a type of software defect that occurs when a program fails to de-allocate memory that is no longer being used. Memory leaks can have serious consequences, potentially leading to such issues as performance degradation or program crashes.

In languages, such as C or C++ that have no built-in garbage collection, a memory leak typically accounts for allocated memory that is no longer reachable via program pointers. Such leaks are often referred to as *physical*. Another type of leak, known as *logical*, refers to unused memory that is still reachable (e.g., through variable references). The focus in the present thesis is on physical leaks only.

A physical leak occurs when a block of memory allocated by a programmer via a memory allocation function (such as `malloc`) is no longer reachable through a pointer that can be used to de-allocate this block (e.g., by using the `free` function). Even though many monitoring techniques for detecting memory leaks have been developed [14, 17, 18, 21–24], they typically report only program locations where leaking blocks have been allocated. While such information provides a good starting point for tracking a memory leak, detecting the location where the leakage actually occurs (and thus where it needs to be eliminated) is not a trivial task. This is especially the case for large programs, where the trace from allocation point to the point of leakage can span across different files and libraries.

Techniques that aim to detect leakage locations have been also reported in the literature. For example, Maebe et al. [20] use machine-level instrumentation to track all pointers to allocated memory using reference count and operations that potentially change the pointer structure of a program and associated locations. At a program point, where reference count for a block drops to zero, the block is considered a memory leak and the program point is determined as the location of the leakage. Clause and Orso [12] track leakage locations in a similar manner via taint analysis, where each tracked block receives a taint mark. The taint marks associated with memory blocks are propagated as execution proceeds. This is based on the propagation policy, which models operations that change memory structure.

Even though the above approaches can identify locations of leakage, they result in a runtime slowdown factor of 100-300. Most importantly, these techniques lack precision. The approach of Maebe et al. reports false alarms and misses memory leaks. While Clause and Orso identify all memory leaks correctly, their propagation policy is neither sound nor complete; that is, the reported sources of the leaked memory are

not guaranteed to be precise.

Another solution that aiming at detection of leakage locations is Boehm GC [25], a conservative garbage collector for C and C++ languages that uses a variation of the mark-and-sweep algorithm. This technique is capable of identifying locations of leakage and is sufficiently scalable to be used with such large projects as Mozilla [26]. However, Boehm GC relies on periodic scanning of the program address space to determine lost memory, and thus also lacks precision in determining locations of leakage.

In summary, even though many techniques for memory leak detection exist, they report only where the leaked memory was allocated. The handful of approaches that target leakage locations specifically result in high overheads or lack precision.

1.1.3 Information Leakage

This section discusses issues associated with the detection of leakage of sensitive information at runtime.

The problem addressed by information leakage detection is to ensure that data (i.e., a set of values in a program run) identified as secret are not exposed externally, for example, through a publicly visible variable, or direct output by a print function. If the secret values in a run of a program are known, the program can be checked for information leakage by calculating dependencies between secret and publicly available data.

A monitoring approach to information flow [13, 27, 28] or taint analysis [29, 30] is appropriate for the detection of information leakage. This is because such an approach captures every assignment and evaluates values for a particular program run. However, tracking every assignment often results in high overhead costs [9, 11, 13].

Additionally, in memory-unsafe languages such as C or C++, precise and scalable information leakage analysis is a challenge due to language features such as pointer arithmetic, weak type system or dynamic memory allocation. As a result, the question of practical information leakage detection in memory-unsafe languages has not been fully addressed by the existing research.

Techniques for information leakage detection often analyse languages that use safe memory models [31–33], where features such as dynamic memory allocation or pointer arithmetic are restricted by the execution environment. Although these approaches have been applied in practice, they cannot be adopted for monitoring C programs.

Other approaches, consider restricting features of target languages. For example, Magazinius et al. [13] developed a framework for inlining dynamic information flow monitors. This approach, however, does not support aliasing and assumes that functions have no side-effects. An approach called Resin [34] tracks values and detects

leakage using data-flow assertions checked at runtime. However, since Resin mainly targets web systems, it does not support analysis in the presence of aliasing or pointer arithmetic.

Assaf et al. [35] described a dynamic flow-sensitive information flow monitor for sound enforcement of non-interference property in programs with pointers. However, this approach uses the semantics of the Clight language [36] (a subset of the C programming language used by the CompCert verified compiler [37]), which does not support pointer arithmetic.

1.2 Aims and Scope

This thesis investigates aspects of monitoring for bug detection that lead to implementations that identify errors precisely and with overheads acceptable for use with testing.

One of the goals of this research is to investigate techniques for detecting memory leaks and locations of leakage in languages where memory is not managed (primarily C and C++). As stated previously, the focus of this approach is to yield a monitoring analysis with overheads acceptable for use with testing. It also concentrates on detecting physical memory leaks only: in other words, detection of logical leaks or tolerating memory leaks [38, 39] (task mainly aimed at eliminating performance degradations) is out of the scope of this work.

This thesis also aims to address the question of high overheads often associated with runtime detection of disclosure of confidential information and support information leakage detection in the presence of features of memory-unsafe languages, such as dynamic memory allocation, pointer arithmetic and aliasing. The scope of this problem is to detect leakage of entire values; detection of values that leak via parts (e.g., bit by bit) is beyond the scope of this thesis. Further, this work concentrates only on leakage detection; it does not involve tracking issues in memory safety, such as buffer overflows or use after free errors.

Finally, this thesis aims to provide generalisations for the monitoring components developed for runtime detection of specific defects. In other words, it aims to employ monitoring primitives identified from the present analysis for memory leaks and information leakage to develop an approach in which monitoring is specified concisely at an abstract level. The aim of such a generalisation is to reduce the development cost of specifying monitoring for defect detection from scratch, but still yield efficient implementations.

In summary, the scope of this thesis is limited to developing monitoring analyses for the detection of memory leaks and information leakage and identifying abstract representations capable of expressing such issues concisely. Thus, monitoring for problems other than bug detection (such as debugging or profiling) or discovering

different defects (e.g., bugs in concurrency) will be beyond the scope of this work.

1.3 Research Questions

The goals stated in the previous section are addressed by instantiating monitoring analyses for specific issues. First, techniques are developed for runtime detection of memory leaks and information leakage; empirical evaluation that implements such techniques is conducted and results reported. Next, a generic monitoring solution is developed that uses abstract descriptions to specify monitoring for software defects concisely.

In summary, the research questions addressed in this thesis can be stated as follows:

1. What features of monitoring enable precise runtime detection of memory leaks and leakage locations, and enable disclosure of confidential information using overheads acceptable for use with testing (as specified on page 3)?

2. What features can be used to specify monitoring using concise abstract specifications that enable instantiation of monitoring analyses?

The following section discusses the contributions made by this thesis.

1.4 Contributions

- This thesis presents an approach to detection of memory leaks in languages that support dynamic memory allocation (e.g., C or C++) with acceptable overheads. The key issue addressed by this technique is the detection of precise locations where the leaked memory was actually lost.

  To enable detection of leakage locations at runtime, the project tracks memory allocated by a program and associate each tracked block with two types of locations: allocation and usage. An allocation location represents a program point at which a memory block associated with it was allocated on the heap. The locations of usage are updated every time a variable containing references to the allocated block is accessed by a running program. This represents the reachability of a memory block with respect to program variables, and is achieved by computing the dereference of a block's address space. Unreachable blocks that have not been de-allocated by the program are reported at the end of execution as memory leaks. Such reports include information that describes where the leaked blocks were allocated and where the leakage occurred. The computation is tunable. Runtime overheads can be reduced for the cost of reporting less information without losing precision.

Further, it is shown that this technique is not limited to the detection of memory leaks: the author employs the monitoring primitives used to detect memory leaks in monitoring analysis for illegal memory modifications.

The approach is supported by a prototype implementation for C programs. Its applicability is demonstrated by monitoring real UNIX utilities and programs selected from the CPU [40] datasets of the Standard Performance Evaluation Corporation (SPEC). The results of this experimentation show that the performance overhead of the present approach compare favourably to those produced by the Valgrind [14] memory debugger.

- This thesis also describes an approach to preventing leakage of sensitive information used by a program at runtime.

  Instead of analysing programs by way of its variables, as is common in information flow or taint analyses, the present technique analyses values. This method has the ability to identify whether a disclosed value represents an information leak with respect to the values considered secret at runtime. Tracking only a handful of values whose disclosure constitutes information leakage reduces the overheads associated with tracking.

  The suggested approach is supported by a prototype implementation for C programs. Its applicability is shown using experimentation on detecting leaks of confidential information in real, security-oriented UNIX software and programs selected from SPEC CPU datasets.

  From experimentation it is shown that the present approach is suitable for addressing application-specific issues, such as the problem of password disclosure. The results of experimentation with a number of UNIX security utilities suggest that the present approach soundly identifies leakage of passwords and incurs overhead of only 1%. Further experimentation shows that the technique is suitable for analysing programs for information flow security vulnerabilities from the Common Weakness Enumeration (CWE). Results of experimentation with real UNIX programs and programs from the SPEC CPU datasets indicate that this approach handles complex programs and yields acceptable overheads.

- Finally, this thesis presents a mechanism called Specification for Monitoring (SFM), for concise and expressive specification of monitoring analysis for a range well-defined problems in bug-checking.

  The strength of SFM is that it separates the semantic issues related to monitoring from their implementation details. This separation of concerns results in compact specifications, as the implementation details are delegated to the implementation of the SFM API, which makes SFM very flexible. Further, although SFM is abstract, it is not so abstract as many specification techniques,

which means that the implementation of the API still has the power to be very efficient.

In addition to the details of abstract representation, SFM also describes the *monitoring API*: a collection of functions that encapsulate monitoring tasks (e.g., tracking source locations). The API relieves users of the burden of dealing with minor implementation details or re-implementing well-known paradigms, yet does not significantly increase overheads compared to manual actions.

To support the present approach, the expressive power of SFM is shown by example, using a case study that demonstrates how well-defined problems in bug checking can be expressed concisely in SFM. The case study presents a complete analysis for such issues as stack overflows, information flow vulnerabilities, resource leakage and SQL injections.

1.5 Thesis Structure

The rest of the thesis is organised as follows.

- Chapter 2 presents a literature review. First, it surveys techniques that concentrate on detection of memory leaks and locations of leakage, and then discusses approaches to dynamic detection of information leakage. The chapter concludes by summarising techniques that enable dynamic analysis using abstract specifications.

- Chapter 3 discusses the syntax, semantics and memory model of an abstract imperative language. Elements of this language are used to explain the semantics of the monitoring analyses presented in Chapters 4 and 5. Chapter 3 also explains the notations and conventions used throughout this document.

- Chapter 4 describes a monitoring technique for the runtime detection of memory leaks and leakage locations. First, this approach is discussed at an abstract level, followed by a discussion of the approximations required to apply it to C programs. Further, this chapter presents the results of the empirical evaluation of the technique, comparing the results of the present research prototype to the results of the Valgrind memory profiler. Finally, an extension for detecting illegal dereferences is presented.

- Chapter 5 presents a monitoring approach to the runtime detection of leakage of the confidential information used by a program. Similar to the monitoring approach discussed in Chapter 4, Chapter 5 first gives an abstract description, then discusses application of this technique to C programs. Finally, the chapter presents the results of experimentation with the prototype implementation of this technique for C programs.

- Chapter 6 summarises the author's experience in monitoring and presents a generic approach to monitoring programs aimed at error detection. This chapter first argues the need for such a generalisation before presenting the SFM language, designed for concise and expressive specification of monitoring at a high level of abstraction. The chapter goes on to demonstrate the applicability of SFM to different problems in error detection using a case study that presents four complete monitoring analyses for detection of stack overflows, information flow vulnerabilities, resource leakage and SQL injections.

- Finally, Chapter 7 gives concluding remarks and discusses directions for future work.

2

Literature Review

This chapter summarises papers that are directly relevant to problems addressed in the body of this thesis (Chapters 4 – 6). The structure of this chapter is therefore as follows: Section 2.1 discusses techniques for the detection of memory leaks, reviewing papers relevant to the authors' approach to detecting memory leaks and leakage locations (see Chapter 4). Section 2.2 summarises the current body of work on the dynamic detection of leakage of sensitive information, reviewing techniques that have similarities with the author's technique to detection of information leakage (see Chapter 5). Finally, Section 2.3 focuses on approaches that enable dynamic analysis using specifications, and discusses work relevant to the technique in monitoring of programs using concise, yet expressive specifications (see Chapter 6).

2.1 Memory Leaks

This section reviews papers directly relevant to the approach to detecting memory leaks and leakage locations presented in Chapter 4 of this thesis. This review concentrates on dynamic methods that enable detection of *physical* memory leaks (i.e., memory no longer reachable via pointers) in languages that support dynamic memory allocation.

The structure of this section is as follows. It first discusses memory debuggers – tools that detect errors by capturing and analysing the memory state of a running program. It then reviews techniques for detecting memory leaks that focuses on identifying the precise locations of leakage. Further, this section summarises leak detection

based on the sampling of program executions. Finally, techniques that detect leaks using hardware features in place of program instrumentations are reviewed.

2.1.1 Memory Debuggers

One of the first tools capable of dynamic detection of memory leaks is *Purify* [21]. *Purify* inserts additional instructions directly into object files monitoring memory allocation and every read or write performed by a program under test. This instrumentation is static (enabled at compile-time). The detection of memory leaks is performed at runtime using a callable garbage detector based on the conventional mark-and-sweep algorithm. Memory blocks identified as no longer referenced by a program are reported as memory leaks.

Alternatively, memory operations can be tracked using dynamic binary instrumentations (DBI). In DBI, an executable (a *client*) is analysed using extra code added to the client at runtime. An example of such analysis for memory leak detection is Memcheck [14] – a memory profiler based on the Valgrind [10] platform. A run of a program under Memcheck first instruments the program with instructions that track memory blocks allocated on the heap (by intercepting calls to memory allocation and de-allocation functions). Before a program terminates, Memcheck reports unreachable blocks that have not been de-allocated as memory leaks. The reachability of a block is decided based on the general purpose registers and data words in the accessible memory of the client.

Dr. Memory [22] also uses DBI instrumentation (via DynamoRIO [41] binary translator) to detect memory leaks. Similar to Valgrind or Purify, Dr. Memory identifies memory leaks based on reachability, such that a heap memory block is considered a memory leak if there exist no pointers to it. At runtime, Dr. Memory detects leaks via a scan that first suspends all threads, and then applies a mark-and-sweep operation to check reachability of the allocated heap blocks. A leak scan is performed at program termination or at a program point specified by the user.

A variety of similar memory debugging and profiling tools capable of detecting memory leaks are available. LeakSanitizer [24] is a memory leak detector integrated into the AddressSanitizer [16] memory error detector. `leaks` [42] is a memory leak detector for Mac OS. This tool periodically scans the memory space of a process and reports allocated but no longer referenced memory buffers as memory leaks. Intel Inspector [23] (built on top of the Pin [43] platform) is a memory error debugger for C, C++, C# and Fortran applications under Windows and Linux; this tool is also capable of detecting memory leaks. Discover [44] is a memory debugger maintained by Oracle. `dmalloc` [45] is a C library of memory management functions that includes facilities for memory-leak tracking and fence-post write detection. `mtrace` [46] is a GNU C library memory debugger that provides essential facilities for tracing memory allocations and reporting non-freed blocks. D.U.M.A. (Detect Unintended Memory

Access) [47] is an open source library for detecting issues such as buffer overflows in C programs; it also provides functionality for reporting allocated but not freed blocks at the end of execution.

Overall, there exist a wide variety of tools suitable for leak detection. The techniques discussed in this section address the detection of memory leaks and report only the leaks' locations of origin (i.e., program locations where lost memory was initially allocated). This can be contrasted with the author's approach, which concentrates on detecting the precise locations where memory was lost. The value of such analysis is demonstrated in Section 4.4.3 (Chapter 4), which compares conventional (allocation only) memory leak reports of Valgrind (via Memcheck tool) to the reports of the author's prototype implementation which additionally contain locations of leakage. The following section discusses techniques that concentrate on similar goals.

2.1.2 Detecting Locations of Leakage

Leakpoint

Clause and Orso [12] developed Leakpoint, a technique that detects sources of memory leaks. Leakpoint tracks memory using dynamic taint analysis. For each allocated memory block Leakpoint creates new a taint mark. A tainted pointer identifies an access alias to that memory block. As execution proceeds, Leakpoint updates taint marks by observing operations on pointers. This uses a *propagation policy* that models each such operation. At runtime Leakpoint keeps track of pointer count per allocated memory block (i.e., taint marks associated with pointers) and identifies leakage locations as those where pointer count has dropped to zero.

The main weakness of Leakpoint is that its propagation policy is neither sound nor complete. That is, while Leakpoint soundly detects leaks, the reported sources of the leaked memory are not guaranteed to be precise. Additionally, the propagation policy does not handle internal scopes. This also may result in reporting spurious leakage locations. Finally, Leakpoint is a DBI approach (built on top of Valgrind) and thus also suffers from high overheads (e.g., Clause and Orso report 300 times runtime overheads). The author's technique addresses similar concerns by using on-the-fly computation rather than the reference count. Because the author uses the points-to information of the program rather the approximation to it (i.e., taint mark propagation) the reported results are precise.

Maebe et al.

Maebe et al. [20] presented a technique that uses machine-level dynamic instrumentation to track all pointers to the allocated memory using reference count. The reference count is computed by monitoring operations that may change the pointer structure of a program. A memory block is deemed to be a memory leak if a memory

operation decreases to zero the reference count associated with that block. In their paper Maebe et al. also discuss the prototype implementation of their approach using the Dynamic Instrumentation, Optimization and Transformation of Applications [8] framework. The empirical evaluation of this technique indicates runtime overheads that range from 200 to 300 times the normal execution.

The downside of Maebe et al.'s approach is that it reports false alarms. This is because this technique tracks only the start addresses of memory blocks; that is, there is no support for handling interior pointers. Further, this approach also misses leaks in the cases where only a part of the pointer is overwritten. This stands in contrast to the author's approach, where such issues are addressed by a traversal of the memory space. This correctly identifies all pointers and their dependencies and identifies leaks soundly. However, the reference count is not, thus requiring a program to terminate in order to detect leaks. Maebe et al.'s technique has no such limitation.

A similar approach, capable of reporting locations of lost memory, was implemented by Meredith [48] in the tool Omega. Clause and Orso [12] indicate that Omega is an independent implementation of the approach suggested by Maebe et al. [20].

Boehm Garbage Collector

Another approach that can potentially provide information, such as locations of leakage, is Boehm GC [25] – a conservative garbage collector for C and C++ languages that uses a variation of the mark-and-sweep algorithm. Boehm GC can also detect memory leaks, although this is not its primary focus.

If used as a leak detector, Boehm GC reports memory blocks that are no longer accessible (i.e., de-allocated in a normal mode of operation) as memory leaks. This, however, relies on periodic scanning of the program address space to determine lost memory, and thus cannot determine precise locations of leakage.

Insure++

Insure++ [49] is a proprietary memory debugger developed by Parasoft. This tool concentrates on runtime analysis and memory error detection for programs implemented in the C and C++ programming languages. Insure++ detects a range of memory errors, including memory leaks. To enable monitoring of memory Insure++ instruments the source code of applications. Similar to the author's approach, Insure++ supports detection of lost memory. However, since Insure++ is a proprietary tool whose implementation details are not publicly available, the present thesis cannot summarise its differences to the author's technique.

2.1.3 Dynamic Sampling

A different approach to dynamic memory leak detection is via sampling executions of code fragments.

Hauswirth and Chilimbi [17] presented a dynamic technique for the runtime detection of memory leaks using bursty tracing [50] (a sampling methodology in continuous program monitoring) and its prototype implementation, called SWAT. SWAT maintains a model of the heap, recording all memory allocations and constantly monitoring all load and store operations sampling executions at a rate inversely proportional to the execution frequency. This results in reports of stale objects (i.e., memory areas that have not been accessed for a particular amount of time) as memory leaks. An advantage of using sampling to detect leaks is that it results in low runtime overheads. From their experimentation (using SPEC benchmarks) Hauswirth and Chilimbi report runtime overheads that are less than 5% compared to unobserved executions. However, due to the application of sampling, which reports memory blocks that have not been accessed for a "long" time as leaks, this technique is known to produce false alarms.

Novark et al. [18] developed an approach to sound runtime detection of memory leaks and bloat in C and C++ applications, called Hound. To detect memory leaks Hound uses a data-based sampling technique. This enables sampling based on access paths to objects. This is different to code-based sampling techniques (e.g., bursty tracing, used by SWAT) that perform sampling based on execution frequency of code. Data-based sampling resolves issues associated with overestimating staleness, and reports no false positives; however it can miss memory leaks. Empirical evaluation of Hound using SPEC CPU benchmarks indicates that its runtime overheads vary from approximately 8% to 102% compared to unobserved executions.

A limitation of Hound is that it can miss errors in cases when a hot (i.e., frequently accessed object) is co-located on a page with a stale object. Lim et al. [19] address this limitation using context aware data sampling, which allocates memory objects using callpaths of allocation sites (context information). The authors demonstrate the benefits of their technique using empirical evaluation that detects memory leaks in benchmarks from the SPEC CINT2000 suite. The results indicate that context aware data sampling detects more memory leaks than conventional data sampling using Hound. However, this does not fully resolve the issues with false negative reports.

One of the most recent approaches to memory leak detection using sampling is Sniper [51]. Sniper concentrates on runtime leak detection via statistical analysis. This employs instruction sampling (via performance monitoring units in commodity processors) to detect staleness of allocated memory. The authors also discuss the results of empirical evaluation using benchmarked code demonstrating that overheads of Sniper are low. This is reflected in an F-measure of 81%.

In sampling-based detection, memory leaks are identified using the notion of *staleness*. In contrast, the author's approach relies on locating points-to relationships. The benefit of the sampling-based techniques is that they can also detect logical leakage (i.e., reachable but unused memory blocks), which is not supported by the author's approach. Therefore, sampling is capable of detecting leaks in memory managed languages; for example, Sleigh [52] detects memory leaks using sampling in Java programs. The author's technique, however, detects all leaks and does not yield false alarms, while sampling-based techniques are known to yield false positive or false negative results.

2.1.4 Detecting Memory Leaks at the Hardware Level

Finally, memory leaks can be detected using facilities provided by an execution environment. Qin et al. [53] presented a tool called SafeMem, which detects memory corruption errors and memory leaks using Error-Correcting Code (an extension of parity memory that can detect single-bit errors) in place of program instrumentation. SafeMem concentrates on detecting *continuous leaks* – memory leaks that result in continuous growth of virtual memory space. Leaks are detected by monitoring the memory usage behaviour and evaluating the life-times of objects with respect to allocations. Empirical evaluation of SafeMem shows runtime overheads of less than 15% of the normal execution. However, due to the nature of the analysis (which makes assumptions on object life-times), SafeMem can report false alarms. MemTracker [54] represents a similar effort that enables memory monitoring and debugging via hardware support.

The author's technique operates at the source level of programs, and therefore is platform and architecture independent.

2.2 Information Leakage

This section reviews related work in the area of dynamic detection of information leakage. It summarises papers directly relevant to the author's approach in detecting of leakage of sensitive information presented in Chapter 5. This focuses on dynamic techniques that address the question of protection of sensitive information against disclosure at runtime.

The structure of this section is as follows. It first reviews techniques in language-based information flow security that detect leakage implicitly, by extending programming languages with security features, rather than explicitly instrumenting programs. This section then discusses related work in the area of data-flow tracking. Similar to the author's approach, these techniques track values during a run of a program using instrumentations. Further, this section summarises research on information flow and taint analysis that track sensitive data in annotated programs. This is followed by a

discussion on techniques that detect information leakage using multiple executions of the program under analysis. Finally, this section summarises papers that address information leakage using testing.

2.2.1 Language-based Information Flow Security

Language-based information flow security [55, 56] enables information analysis in security-typed languages (such as JFlow [57], JIF [58] or FlowCaml [59]) where data types are augmented with annotations that specify security policies for the use of the data at runtime. The specified policies are enforced during a type-checking phase at compile-time. Adding such annotations, however, is a non-trivial task, especially given their semantics, which require extending the target language with the security-oriented type system. The annotations used by the author's approach are only to identify memory locations and values that need to be protected against disclosure.

2.2.2 Data Flow Tracking

Data flow tracking is an alternative approach to detecting information leakage. In contrast to information flow security, which extends the functionality of languages with security features, data flow tracking enables detection of leakage using instrumentations that capture different aspects of leakage. This section now discusses related work in the area of dynamic data flow tracking.

Resin [34] tracks values and detects leakage using data-flow assertions checked at runtime. This approach is similar to that of the present work. Resin targets analysis of web systems and supports specification of policies and filters. However, because Resin mainly supports memory safe languages, such as PhP or Python, this technique tracks only secret values, but not safe locations and does not handle aliasing. A significant limitation of Resin is the need to modify the interpreter to handle security policies. The author's approach modifies only the input program and standard tools (such as `gcc`) can still be used. Further, Resin incurs high overheads (over 400% for some SQL related operations).

LeakProber [60] also addresses information leakage by analysing the flow of data in a program. LeakProber integrates static analysis and runtime tracking to generate a data propagation graph that captures various aspects of the leakage of sensitive information. This differs from the author's approach technique which uses only dynamic analysis. The main aim of LeakProber is to identify vulnerabilities by comparing normal and insecure data propagation graphs. LeakProber also focuses on data that crosses the user/kernel boundary. To achieve this, the authors of LeakProber patch and recompile the kernel to support profiling.

2.2.3 Information Flow Analysis

Information leakage can also be detected using information flow analysis, which tracks the flow of data with respect to the security levels of variables that may only point to the actual data.

Le Guernic et al. [61, 62] analyse information flow to enforce non-interference in sequential and concurrent programs using a combination of dynamic and static analyses. They use the results of static analysis at runtime to detect indirect flows, which has the usual issues with static analysis, such as false positives. Further, the technique by Le Guernic et al. does not handle pointers.

Assaf et al. [35] describe a dynamic flow-sensitive information flow monitor for sound enforcement of the non-interference property in programs with pointers. Assaf et al. formalise a hybrid information flow monitor for a simple imperative language with aliasing. This approach uses the semantics of the Clight language [36] (a subset of the C language used by the CompCert verified compiler [37]), which does not support pointer arithmetic. The present author's approach has no such limitation.

Magazinius et al. [13] inject monitors at the source code level. These monitors are similar to the assertions the author's approach inserts into the programs. Magazinius et al.'s technique handles code evaluated on the fly (i.e., executing strings as code), but it does not handle pointers (or aliasing) and also assumes that the functions have no side-effects. The overheads of their approach range from 20% to 1700%.

Chandra and Franz [63] present a framework for information flow tracking in the Java Virtual Machine. Their approach combines static and dynamic techniques. A static analyser adds annotations, which at runtime are used to update the labels of variables and enforce the security policy currently in place. The key feature of this approach is that it uses completely dynamic policies that can be changed during runtime. Their annotation mechanism relies on static analysis assigning the security level. Chandra and Franz also present experimental results, indicating runtime overheads that vary from 23% to 159%, however it is not clear how much data were tracked. In contrast, program annotations in the present author's approach only classify memory locations as safe or unsafe. Additionally, the author's approach does not need to track or compute security levels of variables in order to soundly identify leakage.

Hedin and Sabelfeld [32] developed a dynamic type system for a subset of JavaScript, incorporating objects, higher order functions, exceptions and dynamic code evaluation. This enforces the property of non-interference, thus protecting programs from leaking private inputs to public outputs. Hedin et al. [33] extend this work and develop a security-enhanced interpreter, called JSFlow, for the full non-strict JavaScript standard (ECMA-262). In contrast to the author's approach JSFlow uses types to detect information leakage: a vulnerability is detected only if all elements can be typed. In the present approach the author uses memory locations and untyped values. However, this approach detects only leaks via explicit information flows. JSFlow

also identifies leakage via implicit flows.

2.2.4 Dynamic Taint Analysis

An alternative approach to information leakage detection is by means of taint analysis in conjunction with DBI; this combination sometimes referred to as dynamic taint analysis. The main benefit of such an approach is its ability to track every bit. This drastically improves the precision of the analysis (since all information is available at runtime) at a cost of runtime overheads of over 50 times [9] the normal execution. Such techniques are described below.

LIFT [29] tracks information flow via dynamic taint analysis by tagging secret values and propagating the tags at execution time. LIFT uses several aggressive optimisation strategies aimed at reducing overheads associated with DBI and information flow tracking. The results of experimentation with LIFT (built on top of the Start-DBT [64] DBI framework) indicate that it reduces runtime overheads by an order of magnitude (compared to a TaintCheck [9] – a taint analysis tool for overwrite attacks detection). The overheads incurred by LIFT average to 3.6 times the normal execution. The advantage of using LIFT is that it does not require source code, and has the ability to track information flow across library calls. This is a limitation of the author's approach which is based on source code instrumentation and thus, requires source code to be available.

TaintDroid [65] is an extension to the Android platform that dynamically tracks the flow of data through third-party applications to identify sensitive information that leaves the system. To solve the issue of high overhead costs associated with tracking of data at the instruction level, this technique combines multiple granularities of tracking: at the variable, message, method and file levels. The authors report overheads of 14% for micro benchmarks. Extensions to TaintDroid have also been proposed [66]. To enable various levels of tracking, TaintDroid requires modification of the runtime environment at the operating system level (i.e., Dalvik virtual machine). This is different to the author's approach, which transforms only the program under analysis. TaintEraser [30] is a similar system for preventing exposure of sensitive information based on dynamic application-level taint analysis (using Pin [43] as its DBI platform). TaintEraser employs object-level propagation, maintains a shadow list of tainted kernel level objects (such as file handles) and monitors changes to these objects. Overall, TaintDroid and TaintEraser lift the granularity of tracking to a higher level (e.g., object or file level) that requires fewer instrumentations and as a consequence reduces the overheads comparing to tracking byte-level tracking. However, this approach leads to imprecise approximations, as byte-level operations are deliberately omitted from the analysis. The author's technique tracks values at memory block level.

2.2.5 Secure Executions

The techniques for detecting information leakage discussed in the previous sections consider single executions of a program instrumented with security checks. This section discusses approaches that detect information leakage via multiple executions of programs.

Capizzi et al. [67] developed a practical approach for preventing information leaks called *shadow execution*. Shadow execution replaces the original program with two copies, such that the first (private) copy is supplied with sensitive information and is prevented from making network connections. The second (public) copy communicates over the network using only non-confidential information, which then can be shared with the private process without loss of confidentiality. Capizzi et al. implemented their approach for the Windows platform and report runtime overheads that range from 23% to 206%. The present author's technique embeds checks for information leakage into the body of a program; under this type of analysis the program never outputs the leaked data, whereas shadow execution exposes them to the sandboxed environment, which needs to be secured. Further, there is an additional cost associated with running a copy of the original program and inter-process communication.

Devriese and Piessens [68] proposed a similar approach for information flow control called *secure multi-execution*, where a program is executed multiple times – once for each security level. This controls a program's public output, produced only if the output matches the appropriate security level. The authors implemented their approach in a model browser. Experimental results indicate that runtime and memory overheads associated with secure multi-execution can be as high as 200%.

Austin and Flanagan [69] have presented a technique for preventing leakage of confidential information and violations of data integrity. In their approach Austin and Flanagan introduce the notion of a *faceted value* – a pair of two raw values that contain both public and private information. By manipulating these values, shadow execution is simulated using a single process. This enables strict information flow control where multi-processing is involved.

2.2.6 Testing

This section summarises papers that combine information leakage detection with testing.

Panorama [70] is an approach to detecting leakage of sensitive information that focuses on tracking information flow under test cases. Panorama performs security information flow analysis in three stages: testing, monitoring and analysing. First, the code under investigation is loaded into the testing environment, where the set of automated tests are conducted and the program behaviour is monitored. The result obtained from monitoring is then analysed against user-defined security requirements.

An alternative approach to information leakage detection is taken by Privacy Oracle [71], which considers an application as a black box. To detect information leaks Privacy Oracle uses a variation of a black box testing technique, where perturbations in the application inputs are mapped to perturbations in the application outputs to discover likely leaks. TightLip [72] is another system that does not require access to the source code of applications. TightLip employs *doppelgangers* – sandboxed copy processes that run in parallel to the original program – and uses divergent process outputs to detect potential leaks.

2.3 Monitoring Specifications

This section summarises techniques that facilitate construction of dynamic analysis using specifications. This review highlights differences between the author's approach to monitoring, called SFM (see Chapter 6) and similar techniques that enable dynamic analysis at a specification level.

The structure of this section is as follows. It first discusses monitoring via instrumentations and then reviews techniques that enable dynamic analysis using streams of events that represent program behaviours. This section further discusses behavioural interface specification languages that enable monitoring using in-line annotations. This is followed by a discussion on techniques in runtime verification that employ high-level abstractions (e.g., formal logic) to describe properties that should hold at execution time. Finally, this section focuses on techniques that use behavioural patterns to observe traces of program events at runtime.

2.3.1 Low-Level Instrumentation

A variety of frameworks that support construction of dynamic analysis have been developed. Early attempts include techniques such as ATOM (Analysis Tools with OM) [73, 74], EEL (Executable Editing Library) [75] and Shade [76]. These frameworks provide infrastructures for code instrumentations at a source code or instruction level. Some similar state-of-the-art solutions include architectures such as LLVM [77], Valgrind [10], Pin [43], DynamoRIO [41] and StarDBT [64]. Such tools typically analyse programs by injecting implementation-specific code at program locations identified by the user. Such an approach provides fine-grained instrumentation functionality (e.g., modify instructions in the target program) for the cost of complex, implementation-level specifications. Since it is the user's responsibility to provide the program points and code for instrumentations, such techniques are considered *manual*. As such, they are beyond the scope of this review which concentrates on approaches that aim to reduce the development overheads involved in specifying dynamic analysis by hand.

2.3.2 Monitoring Program Events

One way to facilitate specification of monitoring is to represent a program run as a stream of events and provide means to observe them, for example, via callback functions scheduled to be executed once specific events are triggered. This section describes monitoring techniques that explore this idea.

BEE++ [78] is a C++ application framework for dynamic analysis of distributed programs based on BEE [79], a system of templates and tools implemented in the C programming language. BEE++ sees program execution as a stream of primitive events. Built on the notion that primitive events should be used to compose more complex events, BEE++ allows events to be extended by way of inheritance. Specifically, an event is encapsulated as a C++ class, and a custom instance of an event is a subclass that inherits from a built-in class provided by the platform. The event processing model of BEE++ consists of the target program, the dynamic analysis tool and the event configuration manager. Events generated by the target program are sent to the dynamic analysis tool, which invokes user-specified code that observes the execution of the events. In addition, the analysis tool generates events and sends them to the target program. This is enabled in order to request additional information required by the analysis.

A Lightweight Architecture for Monitoring (Alamo) [80–82] is a framework for monitoring programs that aims to reduce the cost of writing monitors. Initially developed for Icon programs [83], Alamo has been extended to support C. Alamo instruments programs using the Configurable C Instrumentation (CCI) [84, 85] tool. The instrumented programs transmit events that represent individual units of behaviour. Typical Alamo events include memory references, heap allocations, program control flow and procedure calls. To reduce the number of generated events Alamo uses event masks that specify sets of events that should be observed by the instrumentation. In response to events generated by a program, Alamo invokes monitors specified as C macros. These macros are expanded by the CCI tool and used to instrument target programs.

Olsson et al. [86] suggested an approach to event-driven debugging called Dalek. Dalek is based on the data-flow view of *primitive* and *high level* program events. Primitive events (along with the supported attributes) are defined and raised by a user (for example, via break points). A high-level event is triggered by a primitive event and specifies the code to be executed. This uses a custom language similar to C. Dalek does not support complex event patterns, and is only capable of invoking high-level events on occurrences of primitive events.

Bates [87] suggested an approach to debugging and monitoring of programs called Event Based Behavioural Abstraction (EBBA). In EBBA, events that express behaviours of programs are defined by the programmer via source code annotations.

These annotations generate event instances at runtime. Behaviour models (otherwise known as event patterns) that need to be monitored are expressed using regular expression grammar, where events are represented using tuples of event types and attributes.

Jahiera and Ducasse [88, 89] suggested specifying monitoring of programs in functional and logic languages using a monitoring primitive, called `foldt` – a variation of the `fold` function used for traversal of lists in logic programs. `fold` is similar to the `map` function over lists, bit has an additional argument called accumulator accessed at each step of the execution. `foldt` is designed to perform similar operations over streams of events at runtime. Jahiera and Ducasse implemented this approach for the Mercury programming language [90].

RoadRunner [91] is a tool that supports rapid prototyping with dynamic analyses for concurrent Java programs. RoadRunner provides an API for communicating with events generated by the monitored program. Construction of an analysis using RoadRunner is limited to specifying handlers for built-in events for concurrency analysis (e.g., non-volatile and volatile memory accesses, lock acquire, release and thread operations). RoadRunner keeps track of threads and memory locations using shadow memory. This is done in such a way that during the execution of the monitored program for each thread there exists a shadow thread (which encapsulates information about the program thread) and for each program variable there exists a shadow variable. The values of the shadow objects are tracked by the monitored program as it executes.

Sofya [92, 93] provides support for monitoring of events in concurrency-aware Java environments. Analysis specifications are enabled using a declarative language called EDL (Event Description Language) that describes events and the ways in which they are monitored. In EDL, observed events are specified using a rule-based system, where larger events are composed of primitive events (e.g., method invocations or field reads or writes). Events are processed using a publish/subscribe event handling system. The EDL specifications are translated to Java bytecode and the original programs are instrumented using the Byte Code Engineering Library [94].

SFM also uses events to observe program behaviours. However, a feature that differentiates SFM from most of the techniques described in this section (with the exception of EBBA) is the use of behavioural patterns, allowing for observations of events that occur in specific sequences. Additionally, the monitoring approaches discussed in this section specify monitoring using the implementation language of the monitored programs. This is different to the present approach that specifies monitoring using an intermediate language that is used to generate implementation-level code.

2.3.3 Behavioural Interface Specification Languages

Behavioural interface specification languages (BISLs) provide formal specifications of intended program behaviours. Such specifications are provided via annotations or language extensions that document the desired behaviours using pre- and post-conditions, invariants and assertions. In dynamic analysis, such in-line specifications are used to generate implementation-level monitors. At runtime, these monitors enforce the specified requirements. This section summarises relevant work in this area.

Hatcliff et al. [95] reviewed techniques in formal behavioural specifications of programs. Their survey provides an overview of the various features of different BISLs and their use for automatic verification of programs.

Anna (Annotated ADA) [96, 97] is a BISL for formal specification of the intended behaviour of ADA programs. Anna specifications are provided in the form of annotations associated with ADA syntax constructs. Anna supports type annotations, which impose constraints on the types in the program, and subprogram annotations, which specify the intended behaviours of the program. A number of techniques for conversion of the formal properties in the Anna language into runtime checks have also been developed [98, 99].

Eiffel [100] is an object-oriented programming language developed by Bertrand Meyer. In Eiffel, the intended behaviours are specified by means of *contracts* that document the interface specifications of program components using such language extensions as pre-conditions, post-conditions and class invariants. This design has since become known as the *design-by-contract* principle.

Larch [101] is a two-tier approach to formal specification of program behaviours. The top tier of Larch consists of a BISL; the bottom tier is called the Larch Shared Language (LSL). The LSL, which describes mathematical vocabulary, specifies a mathematical model and the BISL, tailored to a specific programming language, describes the interface and the behaviour of the program. Various BISLs for the top tier have been considered: Larch/C++ [102], Larch/Ada [103], Larch/Smalltalk [104].

Spec# [105] is a behavioural specification language for the .NET platform. Spec# is a superset of the C# programming language enriched with constructs that capture the programmer's intentions. Spec# uses *contracts* that specify how data and methods should be used. The Spec# compiler converts the additional constructs into runtime checks that enforce specifications. Additionally, Spec# employs the Boogie static verifier, which has the ability to check the program statically against the specified requirements.

Java Modelling Language (JML) [106] specifies the intended behaviours of Java classes and interfaces as source code annotations. In JML, behaviours can be described using pre- and post-conditions. Additionally, JML allows assertions to be

placed in the Java code. Burdy et al. [107] survey some of the well-known applications of JML in different areas of program analysis. Bytecode Modelling Language [108] is a similar effort that uses bytecode-level annotations.

E-ACSL [109, 110] is one of the most recent developments in the area. E-ACSL is a subset of ACSL [111], a formal specification language for C programs used by the Frama-C [112] framework. ACSL is based on first-order logic that combines the use of pure (side-effect free) C expressions and keywords that allow for reasoning about the results of functions. Further, ACSL can express most of the functional properties of C programs, and implements the design-by-contract principle, such that a contract can be associated with a function in a program and specify pre- and post-conditions. While ACSL has been designed for static analysis of C programs, E-ACSL was adopted for dynamic analysis specifications. The annotations in E-ACSL are translated into executable monitoring code and embedded into programs as runtime-checks via the E-ACSL2c compiler.

In the present approach definitions of analysis and source code of monitored programs are kept separate. The key difference of SFM and a BISL approach is that a SFM monitoring specification is independent of the monitored program, whereas a BISL specification is embedded in the body of the monitored program. Such separation of concerns offers the benefit of using the same analysis, once defined, for monitoring many programs. SFM, however is less suitable for the specification of application-specific properties. For example, SFM cannot specify pre- and post-conditions or reason about the behaviours of individual variables or objects.

2.3.4 Runtime Verification

Runtime verification is an area of dynamic analysis that focuses on checking program executions against properties provided via a requirement specification. Such specifications are often given via high-level abstractions used to generate observers that monitor the execution of a running program. Checking program runs is enabled either on-line (during the execution of a program) or off-line (by extracting a program trace and verifying it against the set of properties given by the specification). Jin et al. [113] presented a comparative table for a number of well-known approaches to runtime verification, briefly surveying properties of the systems such as target language, scope of analysis, logic used in writing requirement specifications and modes of execution (i.e., on-line or off-line). This section surveys some of the relevant runtime verification techniques.

Monitoring Oriented Programming (MOP) [114] is an approach to runtime verification of programs aimed at validating formalised program requirements at runtime. MOP automatically synthesises monitoring code from higher-level formal requirements specified via source-code annotations. Formalisms that capture program requirements are specified as plug-ins; that is, MOP is independent of any specific

formalism. The MOP technique is implemented in a runtime verification tool called Java-MOP [115]. Similar to the BISL techniques (sometimes classified as runtime verification techniques) MOP targets detection of application-specific requirements via in-line annotations. This is different to the author's approach, in which specifications are independent of the source code of programs.

Program Trace Query Language (PTQL) [116] is an approach to runtime verification that specifies requirements using a SQL-like language called Partiqle. Partiqle uses relational queries over traces of program events viewed as sets of records with associated timestamps. PTQL enables on-line analysis via external specifications and uses state machines to execute queries. Since PTQL is, in essence, a query language, it has expressiveness limitations. For example, the authors of PTQL indicate that it can express issues, such as mismatched method pairs or serialisation errors, but it cannot support analysis for SQL injections in its full generality. Unlike PTQL, SFM is a procedural language that associates procedural routines to be executed when the match is achieved. This has the power to express arbitrary computations.

MaC [117–119] is a framework for monitoring programs against system requirements. MaC specifications are implemented using two languages: Primitive Event Definition Language (PEDL) defines events observed by the system and Meta Event Definition Language (MEDL) specifies formal requirements using PEDL events. Specifications in these languages are used to instrument programs with code that emit program events at runtime and enables dynamic checking of properties given via the specifications. MaC mainly targets application-specific requirements and reasons at the level of specific variables or objects.

Java PathExplorer [120] (JPaX) is a runtime verification tool developed at NASA Ames Research Center. JPaX first extracts relevant events from the executing program, then passes the observer that enables analysis based on a specification that uses temporal logic (via the Maude specification language [121]). Additionally, JPaX enables built-in analyses to detect data race vulnerabilities and deadlocks. Similar solutions using temporal logic to specify requirements include TemporalRover [122] (combination of Linear Temporal Logic and Metric Temporal Logic), Hawk [123] (rule-based temporal logic called Eagle [124]), RuleR [125] (primitive conditional rule-based system) J-LO [126] (Linear-Time Temporal Logic extended with free variables) and jUnit^{RV} [127] (Linear Temporal Logic on finite traces [128]).

A SFM monitor is specified using actions and patterns. This is different to runtime verification, where both components are given in higher-order logic as a single property. Such properties are known to be more compact, yet are not trivial to specify, and are hard to optimise due to the gap between abstract property descriptions and implementation-level monitors generated from the properties. SFM provides a more intuitive way of monitoring specifications and tuning performance: the user specifies *what* behaviours to observe (via patterns) and *how* to observe them using actions.

SFM specifications are also abstract, but they provide a traceable link between abstract patterns and actions and the implementation-level monitors generated from the specifications.

2.3.5 Trace Monitors

Trace monitors originate from aspect oriented programming (AOP) [129, 130] as a generalisation of applying advice (i.e., extra code) on *pointcuts* (collections of well-defined program points).

In traditional AOP, pointcuts refer only to a *current* state. Douence et al. [131] extended the pointcut language and presented stateful, trace-based aspects based on execution history. The authors argued that such aspects are more expressive because they allow for tracking and expressing relationships between events occurring at various points of a program's execution. This model is based on a monitor observing and weaving execution traces, where aspects are defined over sequences of executable states and the aspect weaving is performed on executions, rather than on program code.

Walker and Viggers [132] developed an approach to trace monitoring based on AOP that extends pointcuts to patterns that supporting the specification of multiple events occurring in sequences. In their paper, Walker and Viggers introduce declarative event patterns that explore the idea of defining pointcuts using features of context free grammars. To demonstrate the benefits of their approach, Walker and Viggers extended AspectJ [130] with declarative pattern specifications called *tracecuts*, and compared their design to the standard features of AspectJ.

Another approach to trace monitoring has been presented by Allan et al. [133]. This technique also uses the idea of history-based executions, introducing a feature called *tracematches*. Tracematches enable matching of events in execution traces using patterns based on regular expressions with free variables. Allan et al. also presented a prototype implementation of their approach as an extension to the *abc* AspectJ compiler [134]. This extension supports constructs that allow tracematches to be defined. Further, Allan et al. investigate issues related to efficient implementations and feasibility of monitors generated using *tracematches* [135].

Stolz and Bodden [136] presented a trace monitoring approach where patterns can be specified using LTL properties over AspectJ pointcuts. It is noted, that this work is closely related to the runtime verification approach discussed earlier. Hui and Riely [137] presented semantics for *temporal aspects* that allow for the definition of pointcuts in terms of events that occurred in the past.

The feature that distinguishes SFM approach from the above trace monitors is the use of an abstract language to specify actions. Research on trace monitoring has been focussed on pattern design, and the existing trace monitors specify actions using the implementation language of a target program. SFM enables specifications of analyses

that the existing trace monitors cannot address (e.g., dependency analysis).

The following discusses trace monitors that have the most similarities with SFM.

Program Query Language (PQL)

Program Query Language (PQL) is a trace monitor for Java programs [138]. PQL focuses on tracking method invocations and accesses of field and array elements in related objects. In PQL, events are represented as code patterns, e.g., typed assignments. In contrast, SFM uses abstract events to specify actions performed by the program (e.g., memory allocation). The approach taken by PQL is convenient for monitoring properties of Java objects; however, it complicates specification for more generic analyses. For example, to encode propagation of tainted data in PQL, one needs to specify all relevant code patterns. In SFM this behaviour is captured by a single `flow` event that addresses all data transitions regardless of the code patterns involved (see SFM specification in Listing 6.4). Also, PQL does not support tracking of conditional jumps and cannot address problems in dependency analysis. For example, PQL cannot encode flow-sensitive information flow analysis similar to that shown via the SFM specification in Listing 6.2.

SFM is capable of expressing memory-related problems; (e.g., stack overflow analysis; see Listing 6.1). Since Java is a memory-safe language such issues are outside of PQL's purview. Encoding problems PQL focuses on, for instance mismatched method pairs or SQL injections, is straightforward in SFM. Similar analyses are shown in Listings 6.3 and 6.4.

Arachne

Arachne [139] provides an aspect language for C that features constructs for quantifications over sequences of events. Events supported by this system capture load and store operations and function calls. Arachne is also capable of expressing memory safety problems; for instance, Douence et al. present an AOP specification for buffer overflow detection.

Likewise, SFM supports memory safety; however, expressing such analyses in SFM is more straightforward and compact. For example, Arachne does not have a monitoring API, and thus tasks such as memory tracking need to be implemented by the user. Additionally, memory tracking with Arachne is limited. For example, this approach cannot track stack memory allocations, and thus analyses for stack overflows (see Listing 6.1 for an example) cannot be specified. SFM does not have such limitations.

Arachne provides a powerful pattern language over event sequences capable of expressing problems similar to the resource leakage shown in Listing 6.3. However, functionality that facilitates the analysis needs to be implemented in C that is considered one of the least expressive languages [140].

The literature review presented in this section summarised papers relevant to monitoring techniques presented by this thesis. This section first discussed related work in the area of memory leak and disclosure of confidential information detection. Further, research that concentrates on enabling monitoring using specifications was presented.

The next section presents a memory model that is used to describe monitoring techniques (Chapters 4 – 6) that form the contributions of the present thesis. The following section also presents notations and explains conventions used in the remainder of this thesis.

3 Preliminaries

This thesis presents monitoring techniques to runtime detection of memory leaks and leakage locations, and disclosure of confidential information. To simplify the presentation, the techniqcal details of the monitoring approaches are given at the level of an abstract imperative language.

This chapter describes the syntax and semantics of a simple abstract imperative language similar to the *While* [141] or *Imp* [142] languages. This language and its semantics are referred to as a *standard* model. To explain the monitoring techniques presented in this thesis the standard model is extended with features that capture required properties. For instance, to describe the problem of memory leaks the language is extended with memory allocations, dereference operations and location labels. Such features allow to track memory blocks allocated by a program, detect leaks and report program locations at which leaking blocks were lost. Further, to prevent disclosure of secret values during a run of a program the standard model is extended with operations on pointers and assertions.

The following describes the abstract language in greater detail. Section 3.1 describes the syntax of the language. Its memory model and operational semantics are discussed in Sections 3.2 and 3.3 of this thesis respectively.

3.1 Syntax

Figure 3.1 presents the syntax of an abstract imperative language using a BNF-like notation.

$$
\begin{array}{lcl}
n & ::= & \mathit{Num} \\
v & ::= & \mathit{Var} \\
e & ::= & n \mid v \mid e \oplus e \mid f(e) \\
c & ::= & \texttt{skip} \mid \mathbf{def}(v) \mid v := e \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c \mid c_1 \,;\, c_2 \\
f & ::= & \mathit{Ident} \triangleq c \mid f_1 \,;\, f_2 \\
P & ::= & f \,;\, e
\end{array}
$$

Figure 3.1: Standard Abstract Language

Expressions e consist of numerals n, variables v, binary expressions $e \oplus e$ (where $\oplus$ is a binary operator) and function calls $f(e)$. Set *Expr* denotes the set of all program expressions.

Commands c (given by the set of all program commands *Comm*) consist of atomic commands `skip`, variable definitions $\mathbf{def}(v)$, assignment commands ($v := e$), conditionals (`if` e `then` c_1 `else` c_2), loops (`while` e `do` c) and sequential composition of commands ($c_1 \,;\, c_2$).

Functions f consist of unique function names (*Ident*) followed by a command (c) or a collection of definitions ($f_1 \,;\, f_2$). Program P is a non-empty sequence of function definitions followed by an expression.

The following section discusses the memory semantics of the abstract language shown in Figure 3.1.

3.2 Memory Semantics

Values and memory addresses are represented using natural numbers given by the set $\mathbb{N}$. A particular state of memory, or memory mapping, is represented by the function $m : \mathbb{N} \to \mathbb{N}$ which maps memory addresses to values; that is, $m(a)$ evaluates to a value to which address $a \in \mathbb{N}$ is mapped in the memory mapping m. The set of all possible memory mappings is denoted $\mathit{Mem} = \mathscr{P}(\mathbb{N} \to \mathbb{N})$, where $\mathscr{P}$ is a powerset operator.

Notation $m[\![a \mapsto k]\!]$ is used to denote a memory mapping that is identical to m except for the value mapped to the address a. That is, given that $m \in \mathit{Mem}$ is a memory mapping, $k \in \mathbb{N}$ is a value and $a \in \mathbb{N}$ is a memory address:

$$
m[\![a \mapsto k]\!](x) = \begin{cases} k, & \text{if } a = x \\ m(x), & \text{otherwise} \end{cases}
$$

Every variable v from the set of variables *Var* has a representation in the memory. This is indicated by the semantic function $\rho : \mathit{Var} \to \mathbb{N}$ that maps variables to memory addresses. That is, $\rho(v)$ (where v is a variable belonging to *Var*) evaluates to some memory address $a \in \mathbb{N}$. This representation is unique; that is, for each distinct pair of

$\mathbf{eval}(n, m)$	$= \mathcal{N}(n)$
$\mathbf{eval}(v, m)$	$= \mathbf{eval}(m(\rho(v)))$
$\mathbf{eval}(e_1 \oplus e_2, m)$	$= \mathbf{eval}(e_1, m) \oplus \mathbf{eval}(e_2, m)$
$\mathbf{eval}(f(e), m)$	$= f(\mathbf{eval}(e, m))$

Figure 3.2: Evaluation of Program Expressions

variables x, y belonging to *Var* their addresses are also distinct (i.e., $\rho(x) \neq \rho(y)$).

3.3 Operational Semantics

This section describes evaluation of expressions and the operational semantics of commands of the abstract language.

3.3.1 Evaluation of Expressions

The evaluation of a program expression $e \in \mathit{Expr}$ is given by the function **eval** : $(\mathit{Expr} \times \mathit{Mem}) \rightarrow \mathbb{N}$, where *Expr* is the set of all program expressions, *Mem* is the set of all memory mappings and $\mathbb{N}$ is the set of all values. That is, the value of expression $e \in \mathit{Expr}$ in some memory mapping $m \in \mathit{Mem}$ is given by $\mathbf{eval}(e, m)$.

The rules for the evaluation of expressions of the abstract language are shown via Figure 3.3. The evaluation of numerals n (represented by the set of numerals *Num*) is given by the semantic function $\mathcal{N} : \mathit{Num} \rightarrow \mathbb{N}$ which maps numerals to values. That is, each numeral $n \in \mathit{Num}$ evaluates to a value given by a natural number (i.e., $\mathcal{N}(n)$). Variables $v \in \mathit{Var}$ are direct mappings from their addresses to values in a memory mapping. That is, the value of a variable v in the memory mapping m is given by function application $m(\rho(v))$ that returns the value that the variable v is mapped to in the memory mapping m. Function calls $f(e)$, which potentially result in side-effects, evaluate to applications of the function f on the evaluated expression (i.e., $\mathbf{eval}(e, m)$). Similarly, evaluation of a binary expression $e_1 \oplus e_2$ is $\mathbf{eval}(e_1, m) \oplus \mathbf{eval}(e_2, m)$, i.e., the application of a binary operator $\oplus$ on expressions e_1 and e_2 evaluated using **eval**.

3.3.2 Operational Semantics of Program Commands

The following discusses the operational semantics of the commands of the abstract language.

Commands of the abstract language are given by the set of program commands *Comm*. The operational semantics of the commands (see Figure 3.3) is defined as a relation $\rightarrowtail$: $(\mathit{Comm} \times \mathit{Mem}) \rightarrow (\mathit{Comm} \times \mathit{Mem})$ on configurations $\langle c : m \rangle$, where $c \in \mathit{Comm}$ is a program command and $m \in \mathit{Mem}$ is a memory mapping.

There is no rule for `skip` because $\langle \texttt{skip} : m \rangle$ is a final configuration.

$$\textit{Def:}\ \frac{}{\langle \mathbf{def}(v) : m\rangle \rightarrowtail \langle \texttt{skip} : m\rangle}$$

$$\textit{Asgn:}\ \frac{}{\langle v := e : m\rangle \rightarrowtail \langle \texttt{skip} : m[\![\rho(v) \mapsto \mathbf{eval}(e,m)]\!]\rangle}$$

$$\textit{Seq}_1\textit{:}\ \frac{\langle c_1 : m\rangle \rightarrowtail \langle c_1' : m'\rangle}{\langle c_1\,;c_2 : m\rangle \rightarrowtail \langle c_1'\,;c_2 : m'\rangle} \qquad \textit{Seq}_2\textit{:}\ \frac{\langle c_1 : m\rangle \rightarrowtail \langle \texttt{skip} : m'\rangle}{\langle c_1\,;c_2 : m\rangle \rightarrowtail \langle c_2 : m'\rangle}$$

$$\textit{If}_1\textit{:}\ \frac{}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : m\rangle \rightarrowtail \langle c_1 : m\rangle}\ (\text{where } \mathbf{eval}(e,m) \neq 0)$$

$$\textit{If}_2\textit{:}\ \frac{}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : m\rangle \rightarrowtail \langle c_2 : m\rangle}\ (\text{where } \mathbf{eval}(e,m) = 0)$$

$$\textit{While:}\ \frac{}{\langle \texttt{while}\ e\ \texttt{do}\ c : m\rangle \rightarrowtail \langle \texttt{if}\ e\ \texttt{then}\ (c\,;\texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip} : m\rangle}$$

Figure 3.3: Operational Semantics of Program Commands

Variable definitions (given by Rule *Def*) do not change memory mapping. Assignments $v := e$, where v is a variable in *Var* and e is an expression in *Expr*, replaces the value mapped to the address of a variable v (given by $\rho(v)$) with the result of evaluation of e, i.e., $\mathbf{eval}(e,m)$ (Rule *Asgn*).

Rules Seq_1 and Seq_2 gives the operational semantics of the sequential composition of statements $c_1\ ;c_2$. Rule Seq_1 shows the case where evaluation of c_1 is incomplete and leads to a new command c_1'. Rule Seq_2 shows the case where c_1 evaluates to `skip` (i.e., evaluation of c_1 is completed in a single step and the next step can proceed with evaluation of c_2).

Rules If_1 and If_2 describe the semantics of conditionals `if` e `then` c_1 `else` c_2. Rule If_1 shows the case where the expression e in the condition of the `if` statement evaluates to a non-zero value (that executes c_1), and Rule If_2 describes the case where e is zero and command c_2 is executed. Finally, Rule *While* shows the semantics for loops that is derived from the rules for evaluation of conditional statements. This rule unfolds a single level of the loop and evaluates it as a conditional. That is, based on the value expression e evaluates to, this either executes command c in the body of the loop and unfolds another level or executes `skip` that terminates the execution of the loop.

4

Detection of Memory Leaks and Locations of Leakage

One of the key questions this thesis aims to address is the detection of memory leaks and locations of leakage using overheads acceptable for use with testing. This chapter addresses this question by describing a monitoring analysis and presenting a tunable approach to the detection of memory leaks that reports the locations in the source code where the leakages occur.

To enable detection of memory leaks and leakage locations at runtime, each tracked memory block is associated with two types of locations: allocation and usage. The allocation locations are assigned when blocks are allocated on the heap. The locations of usage are updated every time a variable containing references to the allocated block is accessed by a running program. This reflects the reachability of the block via program variables and is achieved by computing the dereference of a block's address space. Unreachable blocks that have not been de-allocated and the associated locations of leakage are reported at the end of execution. This computation is tunable. Runtime overheads can be reduced, with the cost of reporting less debugging information without losing precision.

The proposed technique is evaluated in an empirical study that uses a prototype implementation for C programs, called Skiff, to analyse real UNIX utilities and programs selected from the SPEC CPU datasets. Experimentation demonstrates that, for the purpose of memory leak detection the overheads of the proposed approach are considerably lower those of the state-of-the-art memory profiler Valgrind [14]. Detection of leakage locations leads to higher overheads; however, this functionality is not

supported by Valgrind.

The details of the presented technique for the detection of memory leaks and leakage locations, and the results of the empirical evaluation previously appeared in a conference publication [143].

This chapter also demonstrates that the set of monitoring primitives used for memory leak detection is sufficient to enable runtime detection issues related to improper use of memory. This is demonstrated via an extension for detecting illegal memory modifications (i.e., modifications of memory locations outside of program allocation). To detect illegal memory modifications, memory information tracked by the memory leak detection technique is reused to check every operation that modifies memory. The results of experimentation with this extension indicate that the overheads of this approach to monitoring for illegal memory modifications also compare favourably with those of Valgrind.

This chapter presents the following contributions:

- A tunable monitoring approach to memory leak detection that identifies locations of leakage.
- An extension for detecting illegal memory modifications.
- A proof-of-concept implementation of the proposed approach, called Skiff.
- An empirical evaluation that compares the results of Skiff to the results produced by a state-of-the-art memory profiler.

The rest of this chapter is organised as follows. Section 4.1 discusses the syntax and semantics of an imperative language used to describe the technique for the detection of memory leaks and locations of leakage at the abstract level. This extends the standard model discussed in Chapter 3, with memory allocation and operations on pointers. Section 4.2 presents a technical description of the proposed approach and Section 4.3 shows how to apply it on C programs. Section 4.4 discusses the empirical results of a prototype implementation. Section 4.5 describes the extension to the memory leak detection technique that aims to support checks for illegal memory modifications. Finally, Section 4.6 offers concluding remarks.

4.1 Syntax and Semantics

The author presents a monitoring approach for the detection of memory leaks and leakage locations using an abstract imperative language (see Chapter 3) extended with dynamic memory allocation and operations on pointers. These extensions allow for definitions and therefore detection of memory leaks. The following section describes the syntax, memory and operational semantics of the extensions used for memory leak detection.

$$
\begin{array}{rcl}
n & ::= & \mathit{Num} \\
l & ::= & \mathit{Lab} \\
v & ::= & \mathit{Var} \\
e & ::= & n \mid l \mid v \mid e \oplus e \mid f(e) \mid \mathbf{deref}(e) \\
c & ::= & \texttt{skip} \mid \mathbf{def}(v) \mid c_1 \,;\, c_2 \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid \\
 & ::= & \langle l : \mathbf{deref}(v) := e \rangle \mid \langle l : v := e \rangle \mid \langle l : v := \texttt{malloc}(e) \rangle \mid \langle l : \texttt{free}(e) \rangle \\
f & ::= & \mathit{Ident} \triangleq c \mid f_1 \,;\, f_2 \\
P & ::= & f \,;\, e
\end{array}
$$

Figure 4.1: Abstract Language Extended with Dynamic Memory Allocation

4.1.1 Syntax

Figure 4.1 presents an abstract imperative language extended with memory allocation and operations for manipulating pointers.

Expression e consists of numerals n, variables v, program labels l, composite expressions $e \oplus e$ (where $\oplus$ is a binary operator), function calls $f(e)$ and dereferences $\mathbf{deref}(e)$.

Command c consists of atomic commands (`skip`), variable definitions ($\mathbf{def}(v)$), conditionals (`if` e `then` c_1 `else` c_2), loops (`while` e `do` c), sequential composition of commands ($c_1 \;; c_2$), labelled assignments $\langle l : v := e \rangle$, and $\langle l : \mathbf{deref}(v) := e \rangle$, where label l (belonging to the set of program labels *Lab*) identifies the source location of the command (e.g., a source code line number) and built-in memory allocation and de-allocation commands $\langle l : v := \texttt{malloc}(e) \rangle$ and $\langle l : \texttt{free}(e) \rangle$, respectively.

Functions f consist of unique function names (*Ident*) followed by a command (c) or a collection of definitions ($f_1 \;; f_2$). Program P is a non-empty sequence of function definitions followed by an expression.

4.1.2 Memory Semantics

This section discusses the semantics of the memory model that acts as an extension to the standard model discussed in Section 3.2.

Let memory blocks span across multiple memory locations. A memory block is denoted by a pair of over the set $\mathbb{N}$ that represents the start and end addresses of the block. That is, a pair $(a, b) \in \mathbb{N}$ represents a memory block such that a is its start address and b is its end address. Let $\mathscr{B} = \mathbb{N} \times \mathbb{N}$ be the set of all blocks. Then, memory allocation is a subset of such pairs. Formally, the set of all possible allocations $\mathscr{A}$ is $\mathscr{P}(\mathscr{B})$ with a typical element denoted by α. A valid allocation is defined as follows.

Definition 1 (Valid allocation) *$\alpha \in \mathscr{A}$ is a **valid** allocation if and only if*

1. *For every $(a, b) \in \alpha$, $a \leq b$.*
2. *For every (a, b) and $(c, d) \in \alpha$, if $(a, b) \neq (c, d)$, then there is no address $i \in \mathbb{N}$, such that $a \leq i \leq b$ and $c \leq i \leq d$.*

That is, in a valid memory allocation $\alpha \in \mathscr{A}$ start addresses of allocated memory blocks are not greater than their end addresses and the allocated blocks are disjoint.

Memory mapping m is the set of pairs $\mathbb{N} \times \mathbb{N}$, where each pair $(a,k) \in m$, where $a,k \in \mathbb{N}$ represents a memory address a mapped to a value k. The set of all possible memory mappings is denoted by the set $Mem = \mathscr{P}(\mathbb{N} \times \mathbb{N})$. That is m is an element of Mem. A valid memory mapping is defined as follows.

Definition 2 (Valid memory mapping) *Given a valid memory allocation $\alpha \in \mathscr{A}$, $m \in Mem$ is a **valid** memory mapping, if and only if for every pair $(a,k) \in m$, where $a \in \mathbb{N}$ is a memory address mapped to a value $k \in \mathbb{N}$, there exists a memory block $(c,d) \in \alpha$, such that $c \leq a \leq d$.*

In other words, in a valid memory mapping $m \in Mem$ every address mapped to a value lies within an allocated block.

Store usage by the program (denoted by σ) is a set of pairs $Var \times \mathbb{N}$, where pair $(v,k) \in \sigma, v \in Var, k \in \mathbb{N}$ represents a variable v bound to a value k. Formally, the set of all possible store usages *Store* is $\mathscr{P}(Var \times \mathbb{N})$, where store usage σ in a particular state is an element of *Store*.

Finally, the set *Lab* denotes the set of all program labels. An element $l \in Lab$ denotes either a defined source location (such as a line number) or an undefined one (denoted $\perp$). Labels are used to track usage of blocks during memory allocation and assignments. The function $\mathbf{loc} : \mathbb{N} \times \mathbb{N} \to Lab$ denotes usage tracking in a particular state. For example, a label associated with a block $(a,b) \in \alpha$, where α is a valid allocation, is retrieved using $\mathbf{loc}(a,b)$. The set of all such functions is denoted Lt (for label tracking).

4.1.3 Operational Semantics

Evaluation of Expressions

The behaviour of program expressions (given by set *Expr*) is defined by the function $\mathbf{eval}(e,\alpha,m,\sigma)$, which evaluates to a value $k \in \mathbb{N}$, where $e \in Expr$ is an expression, α is a memory allocation, m is a memory mapping and σ is a store usage by the program. That is, $\mathbf{eval}(e,\alpha,m,\sigma) = k$ denotes an expression $e \in Expr$ that evaluates to a value $k \in \mathbb{N}$, where $\alpha \in \mathscr{A}$ is a valid allocation, $m \in Mem$ is a valid memory mapping and $\sigma \in Store$ is a store usage by program.

The semantics of the evaluation of expressions is given in Figure 4.2. The evaluation of numerals is given by the semantic function $\mathscr{N} : Num \to \mathbb{N}$, which maps numerals $n \in Num$ to natural numbers. The evaluation of program labels (given by the set *Lab*) is defined using the semantic function $\mathscr{L} : Lab \to \mathbb{N}$ that maps program labels to values from $\mathbb{N}$. That is, each label $l \in Lab$ evaluates to a natural number via $\mathscr{L}(l)$. Evaluation of variables is given by a store usage of a program $\sigma \in Store$. A

$$
\begin{array}{ll}
\textbf{eval}(n,\alpha,m,\sigma) & = \mathcal{N}(n) \\
\textbf{eval}(l,\alpha,m,\sigma) & = \mathcal{L}(l) \\
\textbf{eval}(v,\alpha,m,\sigma) & = k \in \mathbb{N} : \exists (v,k) \in \sigma \\
\textbf{eval}(\textbf{deref}(e),\alpha,m,\sigma) & = \begin{cases} k \in \mathbb{N} & \text{if } \exists (a,k) \in m \wedge a = \textbf{eval}(e,\alpha,m,\sigma) \\ 0 & \text{otherwise} \end{cases} \\
\textbf{eval}(e_1 \oplus e_2,\alpha,m,\sigma) & = \textbf{eval}(e_1,\alpha,m,\sigma) \oplus \textbf{eval}(e_2,\alpha,m,\sigma) \\
\textbf{eval}(f(e),\alpha,m,\sigma) & = f(\textbf{eval}(e,\alpha,m,\sigma))
\end{array}
$$

Figure 4.2: Evaluation of Expressions

variable $v \in \mathit{Var}$ evaluates to the value $k \in \mathbb{N}$ to which it is mapped in the store usage by program, i.e., there exists a pair (v,k) in store σ. Evaluation of dereference expressions $\textbf{deref}(e)$ (where e is a program expression) is given by the memory mapping m. An expression $\textbf{deref}(e)$ evaluates to some value k that is mapped to an address given by the result of the evaluation of e in the memory mapping m. For the case where the address given by e does not exist in the memory mapping m, $\textbf{deref}(e)$ evaluates to a zero value. The evaluation of binary expressions and function calls is standard and has been discussed in Section 3.3.1.

Operational Semantics of Commands

The operational semantics of the commands of the abstract language (shown in Figure 4.3) is defined as a relation $\rightarrowtail$ on configurations: $\langle c : \alpha, m, \sigma, \textbf{loc}\rangle$ and *fault*, where c is a program command, $\alpha \in \mathscr{A}$ is a memory allocation, $m \in \mathit{Mem}$ is a memory mapping, $\sigma \in \mathit{Store}$ is a store usage by the program and $\textbf{loc} \in \mathit{Lt}$ is a function that identifies labels associated with allocated memory blocks. *fault* is a special configuration that indicates an abrupt program termination due to a runtime error.

Command $\langle l : v := \texttt{malloc}(e)\rangle$ (see Figure 4.3 for operational semantics, in Rule *Malloc*) allocates a new memory block (e.g., a sequence of contiguous memory cells of size specified by expression e) and binds the address of the first cell in the allocated segment to a variable v. The allocated block (i.e., added to the memory allocation α) is given by the pair of memory addresses $(a, a + \textbf{eval}(e,\alpha,m,\sigma))$, where a is a newly generated address. The new block added to the memory allocation does not violate the validity of the memory allocation (given in Definition 1). That is, the blocks in the updated memory allocation α^* remain disjoint and the end address of the new memory block is greater than its start address. Additionally, the rule for $\langle l : v := \texttt{malloc}(e)\rangle$ updates aliasing (indicated by the change to the store usage by program σ) such that the value of the variable v in the store usage is updated to the start address of the newly allocated block. That is, v is set to point to the new block in memory allocation α. Finally, Rule *Malloc* associates the label l with the new block (shown via updated label tracking function $\textbf{loc}^*$).

Command $\langle l : \texttt{free}(e)\rangle$ de-allocates a memory block, whose first address is given by expression e. Figure 4.3 describes two rules for command $\langle l : \texttt{free}(e)\rangle$ – Free_1

Malloc: $\dfrac{}{\langle\langle l : v := \texttt{malloc}(e)\rangle : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle \texttt{skip} : \alpha^*, m, \sigma^*, \mathbf{loc}^*\rangle}$

where:

$$\alpha^* = \alpha \cup \{(a, a + \mathbf{eval}(e, \alpha, m, \sigma))\}, \text{ where } a \in \mathbb{N} \text{ is such that}$$
$$\forall x \in \mathbb{N} : a \leq x \leq a + \mathbf{eval}(e, \alpha, m, \sigma),$$
$$\nexists (c, d) \in \alpha : c \leq x \leq d$$
$$\sigma^* = \sigma \setminus \{(w, k) \mid (w, k) \in \sigma \wedge w = v\} \cup \{(v, a)\}$$
$$\mathbf{loc}^*(c, d) = \begin{cases} l & \text{if } (c, d) = (a, a + \mathbf{eval}(e, \alpha, m, \sigma)) \\ \mathbf{loc}(c, d) & \text{otherwise} \end{cases}$$

Free$_1$*:* $\dfrac{}{\langle\langle l : \texttt{free}(e)\rangle : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle \texttt{skip} : \alpha^*, m^*, \sigma, \mathbf{loc}^*\rangle}$
$(\exists (a, b) \in \alpha : a = \mathbf{eval}(e, \alpha, m, \sigma))$

$$\alpha^* = \alpha \setminus \{(a, b)\}$$
$$m^* = m \setminus \{(i, k) \mid a \leq i \leq b\}$$
$$\mathbf{loc}^*(c, d) = \begin{cases} \mathbf{loc}(c, d) & \text{if } (c, d) \neq (a, b) \\ \bot & \text{otherwise} \end{cases}$$

Free$_2$*:* $\dfrac{}{\langle\langle l : \texttt{free}(e)\rangle : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \mathit{fault}}$ $(\nexists (a, b) \in \alpha : a = \mathbf{eval}(e, \alpha, m, \sigma))$

VarAsgn: $\dfrac{}{\langle\langle l : v := e\rangle : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle \texttt{skip} : \alpha, m, \sigma^*, \mathbf{loc}^*\rangle}$

where:

$$\sigma^* = \sigma \setminus \{(w, k) \mid (w, k) \in \sigma \wedge w = v\} \cup \{(v, \mathbf{eval}(e, \alpha, m, \sigma))\}$$
$$\mathbf{loc}^*(a, b) = \begin{cases} l & \text{if } (a, b) \in R_v^+(\alpha, m, \sigma, v) \\ \mathbf{loc}(a, b) & \text{otherwise} \end{cases}$$

MemAsgn$_1$*:* $\dfrac{}{\langle\langle l : \mathbf{deref}(v) = e\rangle : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle \texttt{skip} : \alpha, m^*, \sigma, \mathbf{loc}^*\rangle}$
$(\exists (a, k) \in m : a = \mathbf{eval}(v, \alpha, m, \sigma))$
where:

$$m^* = m \setminus \{(a, b) \mid (a, b) \in m \wedge a = \mathbf{eval}(v, \alpha, m, \sigma)\}$$
$$\cup \{\mathbf{eval}(v, \alpha, m, \sigma), \mathbf{eval}(e, \alpha, m, \sigma)\}$$
$$\mathbf{loc}^*(a, b) = \begin{cases} l & \text{if } (a, b) \in R_v^+(\alpha, m, \sigma, v) \\ \mathbf{loc}(a, b) & \text{otherwise} \end{cases}$$

MemAsgn$_2$*:* $\dfrac{}{\langle\langle l : \mathbf{deref}(v) = e\rangle : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \mathit{fault}}$ $(\nexists (a, k) \in m : a = \mathbf{eval}(v, \alpha, m, \sigma))$

Figure 4.3: Operational Semantics of Program Commands

$$\textit{Def:}\ \frac{}{\langle \mathbf{def}(v) : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle \mathtt{skip} : \alpha, m, \sigma \cup \{(v,0)\}, \mathbf{loc}\rangle}$$

$$\textit{Seq}_1\textit{:}\ \frac{\langle c_1 : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle c_1' : \alpha', m', \sigma', \mathbf{loc}'\rangle}{\langle c_1 \,;\, c_2 : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle c_1' \,;\, c_2 : \alpha', m', \sigma', \mathbf{loc}'\rangle}$$

$$\textit{Seq}_2\textit{:}\ \frac{\langle c_1 : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle \mathtt{skip} : \alpha', m', \sigma', \mathbf{loc}'\rangle}{\langle c_1 \,;\, c_2 : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle c_2 : \alpha', m', \sigma', \mathbf{loc}'\rangle}$$

$$\textit{If}_1\textit{:}\ \frac{}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle c_1 : \alpha, m, \sigma, \mathbf{loc}\rangle}$$

(where $\mathbf{eval}(e, \alpha, m, \sigma, \mathbf{loc}) \neq 0$)

$$\textit{If}_2\textit{:}\ \frac{}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \langle c_2 : \alpha, m, \sigma, \mathbf{loc}\rangle}$$

(where $\mathbf{eval}(e, \alpha, m, \sigma, \mathbf{loc}) = 0$)

$$\textit{While:}\ \frac{}{\begin{array}{l}\langle \mathtt{while}\ e\ \mathtt{do}\ c : \alpha, m, \sigma, \mathbf{loc}\rangle \rightarrowtail \\ \qquad \langle \mathtt{if}\ e\ \mathtt{then}\ (c \,;\, \mathtt{while}\ e\ \mathtt{do}\ c)\ \mathtt{else}\ \mathtt{skip} : \alpha, m, \sigma, \mathbf{loc}\rangle\end{array}}$$

Figure 4.3: Operational Semantics of Commands (cont.)

and *Free*$_2$. Rule *Free*$_1$ shows the case where an expression e supplied as an input to function `free` evaluates to the start address of one of the memory blocks in allocation (indicated by the side condition in Rule *Free*$_1$). For this case function call `free`(e) removes the block whose start address is given by the result of evaluation of expression e from memory allocation (indicated by the updated memory allocation α^*). Additionally, this removes all memory mappings belonging to the freed block from the memory mapping m (indicated by the updated memory mapping m^*) and removes the label associated with the freed block from label tracking (given by the updated function $\mathbf{loc}^*$). Rule *Free*$_2$ describes the case where the input to the function `free` is invalid, i.e., it does not correspond to the start address of an allocated memory block (indicated by the side condition in Rule *Free*$_2$). In this case the call to `free` leads to abrupt program termination (indicated by the special configuration *fault*).

Rules *VarAsgn*, *MemAsgn*$_1$ and *MemAsgn*$_2$ in Figure 4.3 show semantics of assignments that update the memory and alias map. Variable assignment (given by Rule *VarAsgn*) updates store usage by program σ. That is, a variable assignment $\langle l : v := e\rangle$ updates the value associated with the variable v to the value given by the result of evaluation of expression e. Assignments via a dereference operator $\langle l : \mathbf{deref}(v) := e\rangle$ (given by Rules *MemAsgn*$_1$ and *MemAsgn*$_2$) update memory mapping. Rule *MemAsgn*$_1$ describes the case where the variable v points to a block from memory allocation (i.e., there exists a memory mapping from the address given by v to a value in the memory mapping m). *MemAsgn*$_1$ associates a new value (given by evaluation of e) with a memory address which value is given by the evaluation of

variable v. Rule *MemAsgn*$_2$ gives semantics for the case where dereference **deref**(v) is invalid (i.e., v points to an address that is not in the memory mapping) which leads to an abrupt program termination via the special configuration *fault*. Additionally, both types of assignments update label tracking (via the updated function **loc***).

Rule *Def* shows the operational semantics for variable definitions **def**(v). This updates store usage by the program by associating the variable with a zero value.

The semantics of the remaining commands (i.e., conditionals `if` e `then` c_1 `else` c_2, loops `while` e `do` c and sequential composition of statements c_1 ; c_2) is standard (see Section 3.3).

Memory Leak

This section formally defines memory leaks. These definitions help in capturing memory leaks and leakage locations at the level of the abstract language.

A variable v points to a memory block (a, b) if the value v is bound to lies within (a, b). This is formally defined as follows.

Definition 3 (Points to via variable) *Given a valid memory allocation $\alpha \in \mathscr{A}$, valid memory mapping $m \in Mem$, store usage by program $\sigma \in Store$, variable $v \in Var$ and allocated memory block $(a, b) \in \alpha$, v is said to* ***point*** *to (a, b) if and only if $a \leq$ **eval**$(v, \alpha, m, \sigma) \leq b$.*

An allocated memory block (a, b) points to another allocated block (c, d) if and only if a memory address within (a, b) is mapped to an address that lies within (c, d). The relation R_b in Definition 4 defines this formally.

Definition 4 (Points to via block) *Given a valid memory allocation $\alpha \in \mathscr{A}$, valid memory mapping $m \in Mem$, store usage by program $\sigma \in Store$, and allocated memory block $(a, b) \in \alpha$, the binary relation:*

$$R_b(\alpha,m,\sigma,(a,b)) = \{(c,d) \mid (c,d) \in \alpha \wedge (\exists i \in \mathbb{N} : a \leq i \leq b \wedge c \leq \textbf{eval}(\textbf{deref}(i), \alpha, m, \sigma) \leq d\}$$

defines the set of memory blocks in α, block (a, b) ***points to****.*

Thus, in allocation α, memory mapping m and store usage σ, block (a, b) points to block (c, d) if and only if $(c, d) \in R_b(\alpha, m, \sigma, (a, b))$.

A given block b_n is accessible from another block b_0 if there exists a sequence of blocks $b_1, \cdots, b_{n-1}$ such that for all i between 0 and $n-1$, b_i points to b_{i+1}. This is formally defined by the relation R_b^+ given in Definition 5.

Definition 5 (Accessibility) *Given a valid memory allocation $\alpha \in \mathscr{A}$, valid memory mapping $m \in Mem$, store usage by program $\sigma \in Store$ and an allocated memory block $(a,b) \in \alpha$, binary relation:*

$$R_b^+(\alpha, m, \sigma, (a,b)) = \{(e,f) \mid (e,f) \in R_b(\alpha, m, \sigma, (a,b)) \vee (\exists (c,d) \in R_b(\alpha, m, \sigma, (a,b)) : (e,f) \in R_b^+(\alpha, m, \sigma, (c,d))\}$$

*defines the set of blocks **accessible** from (a,b).*

Thus, for a memory allocation α, memory mapping m and store usage σ, a memory block $(c,d) \in \alpha$ is accessible from block $(a,b) \in \alpha$ if and only if $(c,d) \in R_b^+(\alpha, m, \sigma, (a,b))$.

A variable $v \in Var$ is said to reference a memory block ((a,b) belonging to a valid memory allocation α) if v points to (a,b), or there exists some block $(c,d) \in \alpha$, such that v points to (c,d) and (a,b) is accessible via (c,d). This is formally defined by the relation R_v^+ in Definition 6.

Definition 6 (Reference) *Given a valid memory allocation α, block $(a,b) \in \alpha$, valid memory mapping m, store usage σ and variable $v \in Var$, binary relation*

$$R_v^+(\alpha, m, \sigma, v) = \{(a,b) \mid (a,b) \in \alpha : a \leq \mathbf{eval}(v, \alpha, m, \sigma) \leq b \vee (\exists (c,d) \in \alpha : c \leq \mathbf{eval}(v, \alpha, m, \sigma) \leq d \wedge (a,b) \in R_b^+(\alpha, m, \sigma, (c,d)))\}$$

*defines the set of blocks **referenced** by variable v.*

Thus, given a memory allocation $\alpha \in \mathscr{A}$, memory mapping $m \in Mem$ and store usage by program $\sigma \in Store$, variable $v \in Var$ references an allocated memory block $(a,b) \in \alpha$, if $(a,b) \in R_v^+(\alpha, m, \sigma, v)$.

Given the above definitions it is indicated that a block (a,b) is a **memory leak** when it is not referenced by any of the program variables. Formally, the definition of the memory leak is as follows.

Definition 7 (Memory leak) *Given a valid memory allocation $\alpha \in \mathscr{A}$, valid memory mapping $m \in Mem$, store usage $\sigma \in Store$, a memory block $(a,b) \in \alpha$, is a **memory leak** with respect to store usage by program σ, if there exists no pair $(v,k) \in \sigma$, $v \in Var$, $k \in \mathbb{N}$, such that $(a,b) \in R_v^+(\alpha, m, \sigma, v)$.*

That is, block (a,b) is not referenced by program variables from the given store usage.

In a particular state of computation (given by memory allocation α, memory mapping m, store usage σ, and label tracking function **loc**), the set of memory leaks is given by the set of blocks not referenced by the program variables. Additionally, each

leaking block is associated with a program label that identifies the source location of the leakage via a label tracking function **loc**. Formally, the definition of memory leaks associated with leakage locations in a particular state of computation is provided via the following definition.

Definition 8 (Memory leaks) *Given a valid memory allocation $\alpha \in \mathscr{A}$, valid memory mapping $m \in Mem$, store usage $\sigma \in Store$, and a label tracking function $\mathbf{loc} \in Lt$, the set of all **memory leaks** is given by the set:*

$$Leaks(\alpha, m, \sigma, \mathbf{loc}) = \{(a, b, l) \mid (a, b) \in \alpha \\ \wedge \nexists (v, k) \in \sigma : (a, b) \in R_v^+(\alpha, m, \sigma, v) \wedge l = \mathbf{loc}(a, b)\}$$

where each triple (a, b, l), $(a, b) \in \alpha$, $l \in Lab$ identifies an allocated block (a, b) leaking at program location l.

4.2 Memory Leak Detection

This section presents technical details of the memory leak detection technique. This technique instruments the program P with statements that track memory allocation and detect memory leaks at runtime. A run of the modified program P' reports any memory leaks that occurred at the end of its execution.

4.2.1 Memory Tracking State

Data structure T_α is used to keep track of the memory state during the execution of the program. A state of T_α describes the state of memory that has been tracked during the execution of the transformed program P'. T_α is given by a set of 4-tuples $\mathbb{N} \times \mathbb{N} \times Lab \times Lab$. The set of all memory tracking states is denoted Mt (for memory tracking) and given by $\mathscr{P}(\mathbb{N} \times \mathbb{N} \times Lab \times Lab)$, that is T_α is an element of Mt. Each element $(a, b, l_a, l_u) \in T_\alpha, a, b \in \mathbb{N}, l_a, l_u \in Lab$ represents a memory block (a, b) (where a and b are its start and end addresses), that was allocated at a program location given by label l_a. Label l_u represents the last known location at which block (a, b) was referenced via a variable. Labels l_a and l_u are referred to as *allocation* and *usage* labels respectively.

A memory tracking state T_α is valid if and only if T_α tracks only allocated memory blocks and each allocated block is tracked exactly once. The notion of validity for a memory tracking state is formalised using the following definition.

Definition 9 (Valid memory tracking state) *Given a valid memory allocation $\alpha \in \mathscr{A}$, a memory tracking state $T_\alpha \in Mt$ is valid if and only if:*

1. $\forall (a, b, l_a, l_u) \in T_\alpha : (a, b) \in \alpha$.

2. $\forall (a,b) \in \alpha, \exists! \, (c,d,l_a,l_u) \in T_\alpha : (c,d) = (a,b)$.

That is, each element of a **valid** memory tracking state T_α tracks an allocated memory block (given by *Clause 1* of Definition 9) and T_α has a unique map for each allocated block (given by *Clause 2* of Definition 9).

4.2.2 Semantics of Monitoring Commands

This section describes the semantics of commands used to instrument an original program (P) with functionality that enables the detection of memory leaks and their associated leakage locations. This instrumentation yields a modified program P'. The monitoring commands are presented as functions that change the memory tracking state T_α as the modified program P' executes. The final memory tracking state (i.e., the state of T_α at the point of termination of P') describes the detected memory leaks and captured locations of allocation and usage.

The following discusses monitoring commands and their operational semantics.

The operational semantics of monitoring commands given by the set $Comm_m$ (Figure 4.4) is defined as a relation $\rightarrowtail_m$ on configurations: $\langle c_m : T_\alpha, m, \sigma \rangle$ where $c_m \in Comm_m$ monitoring command, $T_\alpha \in Mt$ is a memory tracking state, $m \in Mem$ is a memory mapping and $\sigma \in Store$ is a store usage by program.

record(T_α, a, b, l)

Function **record** (shown via Rule *Record*, Figure 4.4) tracks an allocated memory block whose start and end addresses are given by arguments a and b and the source location of the allocation is given by label l. Given a set $T_\alpha \in Mt$ and memory address $a, b \in \mathbb{N}$ and a label $l \in Lab$ a call to **record** adds an element (a,b,l,l) to the memory tracking state T_α. Note, that the usage label (which describes the last known reference to the block) is also set to l. This is because this command is designed to record memory blocks allocated via a program command $\langle l : v \coloneqq \texttt{malloc}(e) \rangle$ that aliases the newly allocated memory block using variable v. Thus, the allocated block is referenced by variable v (see Definition 6).

delete(T_α, a)

Function **delete** (shown via Rule *Delete*, Figure 4.4) represents a de-allocation of an allocated memory block. Given a set $T_\alpha \in Mt$ and a memory address $a \in \mathbb{N}$ **delete** removes an element from T_α, whose start address is a.

updateLabel$(T_\alpha, v, l, mode)$**.**

updateLabel (shown via Rules $UpdateLabel_1$, $UpdateLabel_2$ and UpdateLabel_3 in Figure 4.4) is the main memory tracking function. The inputs to this function are a set

Record: $$\frac{}{\langle \mathbf{record}(T_\alpha, a, b, l), T_\alpha, m, \sigma\rangle \rightarrowtail_m \langle \mathtt{skip}, T_\alpha{}', m, \sigma\rangle}$$

where $T_\alpha{}' = T_\alpha \cup \{(a, b, l, l)\}$

Delete: $$\frac{}{\langle \mathbf{delete}(T_\alpha, a), T_\alpha, m, \sigma\rangle \rightarrowtail_m \langle \mathtt{skip}, T_\alpha{}', m, \sigma\rangle}$$

where $T_\alpha{}' = T_\alpha \setminus \{(c, d, l_a, l_u) \mid (c, d, l_a, l_u) \in T_\alpha \wedge c = a\}$

UpdateLabel$_1$: $$\frac{}{\langle \mathbf{updateLabel}(T_\alpha, v, l, mode), T_\alpha, m, \sigma\rangle \rightarrowtail_m \langle \mathtt{skip}, T_\alpha{}', m, \sigma\rangle}$$
($mode = 0$)

where $A = \{(a, b) \mid (a, b) \in T_\alpha\}$
$R = R_v^+(A, m, \sigma, v)$
$T_\alpha{}' = T_\alpha \setminus \{(a, b, l_a', l_u') \mid (a, b) \in R\}$
$\cup \{(a, b, l_a, l') \mid (a, b) \in R \wedge (a, b, l_a, l_u) \in T_\alpha \wedge l' = l\}$

UpdateLabel$_2$: $$\frac{}{\langle \mathbf{updateLabel}(T_\alpha, v, l, mode), T_\alpha, m, \sigma\rangle \rightarrowtail_m \langle \mathtt{skip}, T_\alpha{}', m, \sigma\rangle}$$
($mode > 0$)

where $A = \{(a, b) \mid (a, b) \in T_\alpha\}$
$R = R_v^+(A, m, \sigma, v)$
$R' = R \setminus \{(c, d) \mid \exists (e, f) \in R : (c, d) \in R_b^+(A, m, \sigma, (e, f)) \wedge e - f > mode\}$
$T_\alpha{}' = T_\alpha \setminus \{(a, b, l_a', l_u') \mid (a, b) \in R'\}$
$\cup \{(a, b, l_a, l') \mid (a, b) \in R' \wedge (a, b, l_a, l_u) \in T_\alpha \wedge l' = l\}$

UpdateLabel$_3$: $$\frac{}{\langle \mathbf{updateLabel}(T_\alpha, v, l, mode), T_\alpha, m, \sigma\rangle \rightarrowtail_m \langle \mathtt{skip}, T_\alpha, m, \sigma\rangle}$$
($mode < 0$)

Report: $$\frac{}{\langle \mathbf{report}(T_\alpha), T_\alpha, m, \sigma\rangle \rightarrowtail_m \langle \mathtt{skip}, T_\alpha, m, \sigma\rangle}$$

Figure 4.4: Operational Semantics of Monitoring Commands

$T_\alpha \in Mt$, variable $v \in Var$, a label $l \in Lab$ and a value $mode \in \mathbb{N}$. The task of **updateLabel** is to update the usage labels of memory blocks that participated in assignments. For example, given an assignment command $\langle l : v := e \rangle$, where v is a variable and e is an expression, **updateLabel** sets usage labels of all memory blocks referenced by v to l. The fourth argument, *mode*, is used to limit the set of blocks whose usage labels are updated. For example, update only a subset of memory blocks referenced by v.

Informally, the execution of **updateLabel** is as follows. Given an input variable v, **updateLabel** identifies the set of memory blocks (say R) referenced by v (i.e., memory blocks that can be accessed through v). R is populated by a recursive walk that dereferences addresses of all memory blocks in R and identifies points-to relationships via information stored in T_α that captures memory allocation. For example, given that v points to a memory block (a, b), which in turn points to some block (c, d), **updateLabel** first dereferences the value of v, then identifies block (a, b) as being pointed to by v and then adds it to R. Further, it dereferences each value in the range $[a, b]$ and adds (c, d) to R (since (a, b) points to (c, d)). It then searches through the range $[c, d]$ and finalises the search (since block (c, d) does not point to any other blocks). Finally, **updateLabel** updates usage labels of elements of the memory tracking state T_α that correspond to blocks in R. For example, given that R contains a memory block (a, b), then some element (a, b, l', l'') of T_α that tracks (a, b) is updated to (a, b, l', l) (where l is an input label). Note that since **updateLabel** recomputes usage labels using memory allocation tracked via T_α, dangling pointers potentially introduced via assignments or memory de-allocation do not affect the precision of label tracking.

The fourth argument of **updateLabel** (*mode*) is used to control the amount of dereferences **updateLabel** function performs. This aims to reduce overheads associated with dynamically computing the points-to information. Based on the value of *mode*, three different modes of computation are identified. In the **full** mode (where the value of *mode* is 0), each block is searched for pointers by dereferencing its contents. In the **partial** mode (where the value of *mode* greater than 0), the dereference is performed only for blocks of a size less than that given by the value of *mode*. For instance, in the example from the previous paragraph, **updateLabel** dereferences ranges $[a, b]$ and $[c, d]$, which correspond to memory blocks (a, b) and (c, d), regardless of their size. This is the behaviour of the **full** mode. Let some value $mode \in \mathbb{N}$ supplied as the fourth argument to **updateLabel** is greater than zero (i.e., **partial** mode) and is such that $(b - a) < mode$ and $(d - c) > mode$. That is, the size of block (a, b) is less than the value of *mode* and the size of block (c, d) is greater. Then, **updateLabel** only searches through the range $[a, b]$ but never dereferences $[c, d]$. Finally, in the **minimal** mode (where the value *mode* is less than zero), no dereferencing is performed and **updateLabel** immediately returns. This mode reduces the analysis to identifying the locations of allocations generated via **record**.

Formally, the result of an execution of **updateLabel** is shown via Rules $UpdateLabel_1$,

UpdateLabel$_2$ and *UpdateLabel*$_3$ in Figure 4.4. Rule *UpdateLabel*$_1$ shows an execution of **updateLabel** in the **full** mode (where the value of *mode* is equal to zero). Set A denotes memory allocation (tracked via T_α) and set R denotes the set of memory blocks referenced via a variable v (see Definition 6). Rule *UpdateLabel*$_2$ shows the execution of **updateLabel** in the **partial** mode. This reduces the set R (which contains all blocks referenced by variable v) to the set R' such that R' contains no memory blocks accessible through blocks which sizes are greater than the value of *mode* (computed via relation R_b^+ given by Definition 5). Finally, Rule *UpdateLabel*$_3$ shows the semantics of an execution of **updateLabel** in the **minimal** mode. In this case, the behaviour of this function is equivalent to `skip` (as no label updates, and therefore changes to the memory tracking state are performed).

report(T_α)

Monitoring function **report**(T_α) (shown via Rule *Report*, Figure 4.4) reports memory leaks based on the state of T_α. For each element (a, b, l_a, l_u) belonging to T_α, the function **report** reports a memory block (a, b) allocated at location l_a and last referenced by a variable at location l_u as a memory leak. This can be shown to be consistent with Definition 8. Note, that the invocation of **report** does not modify the memory tracking state.

4.2.3 Syntactic Transformations

This section discusses transformations (see Figure 4.5) that are used to generate a modified program P', such that P' is equipped with statements that track memory allocations and detect memory leaks and leakage locations.

Initialisation

The first step of the transformations instruments an input program P, with a data structure T_α to keep track of the memory state and a global variable *mode* (see Figure 4.5, Rule *Program*). *mode* is a user-supplied value (indicated by the statement $\langle l : mode := \langle \texttt{INPUT} \rangle \rangle$, which allows to control over the behaviour of the **updateLabel** function.

Memory Allocation

Tracking of memory allocations is enabled via Rules *Malloc* and *Free*. Each statement that allocates memory (i.e., $\langle l : v := \texttt{malloc}(e) \rangle$) is followed by a call to **record**($T_\alpha, v, v+e, l$), where v evaluates to the start address of the allocated block, expression $v+e$ evaluates to its end address and l is a block's allocation label. **record** captures addresses of allocated blocks and locations of their allocation and records them to a memory

Def: $\dfrac{}{\mathbf{def}(v) \rightsquigarrow \mathbf{def}(v)}$ *Skip:* $\dfrac{}{\texttt{skip} \rightsquigarrow \texttt{skip}}$

If: $\dfrac{c_1 \rightsquigarrow c_1',\ c_2 \rightsquigarrow c_2'}{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'}$

While: $\dfrac{c \rightsquigarrow c'}{\texttt{while}\ e\ \texttt{do}\ c \rightsquigarrow \texttt{while}\ e\ \texttt{do}\ c'}$

Malloc: $\dfrac{}{\langle l : v := \texttt{malloc}(e)\rangle \rightsquigarrow \langle l : v := \texttt{malloc}(e)\rangle;\ \mathbf{record}(T_\alpha, v, v+e, l);}$

Free: $\dfrac{}{\texttt{free}(e) \rightsquigarrow \mathbf{delete}(T_\alpha, e);\ \texttt{free}(e);}$

VarAsgn: $\dfrac{}{\langle l : v := e\rangle \rightsquigarrow \langle l : v := e\rangle;\ \mathbf{updateLabel}(T_\alpha, v, l, mode);}$

MemAsgn: $\dfrac{}{\langle l : \mathbf{deref}(v) := e\rangle \rightsquigarrow \langle l : \mathbf{deref}(v) := e\rangle;\ \mathbf{updateLabel}(T_\alpha, v, l, mode);}$

Function: $\dfrac{c \rightsquigarrow c'}{f \triangleq c \rightsquigarrow f \triangleq c'}$

Program: $\dfrac{\tilde{f} \rightsquigarrow \tilde{f}'}{f;\ e \rightsquigarrow \mathbf{def}(T_\alpha);\ \mathbf{def}(mode);\ \langle l : mode := \langle \texttt{INPUT}\rangle\rangle\ ;\ f';e\ \mathbf{report}(T_\alpha)}$

Figure 4.5: Syntactic Transformations

tracking structure T_α immediately after the blocks are allocated via `malloc`. Rule *Free* inserts calls to **delete**(T_α, e) before calls to `free`(e), which de-allocates memory. This stops the tracking of memory blocks de-allocated by the program by deleting them from the memory tracking structure T_α.

Label Tracking

Each assignment statement is appended with a call to the function **updateLabel** (see Figure 4.5, Rules *MemAsgn* and *VarAsgn*), which tracks assignments of memory blocks referenced by variables. Calls to **updateLabel** update usage labels. At any given state, a usage label associated with a block indicates a source location at which that block was last known to be accessible via a variable.

In the present approach **updateLabel** is the main cause of the runtime overhead. To reduce overheads, the behaviour of **updateLabel** is controlled externally via the fourth argument of **updateLabel** – global variable *mode*. Thus the present approach results in a tunable tool where the user can determine an acceptable level of overheads. The present approach supports three modes of execution: **minimal**, **partial** and **full**. In the *minimal* (where the value of *mode* is less than zero), **updateLabel** does not track usage labels. Thus information collected is limited to the existence of memory leaks and the locations of their allocation. In the **full** mode of (where the value of *mode* is 0), each block (regardless of its size) is searched for pointers. This may result in larger overheads, however it allows for the identification of all usage labels. In the **partial** mode (where the value of *mode* is greater than zero) only blocks of size strictly less than *mode* value are traversed. This allows the overheads to be reduced due to traversal of memory blocks of particular sizes only. For example, large memory blocks that are considered data only and do not contain any pointers are skipped.

Memory Leak Reporting

The final stage of the instrumentation injects a call to a reporting function before the program exits, such that **report**(T_α) is the last executed statement (see Figure 4.5, Rule *Program*). Thus, at the point of execution of a reporting function, no other memory operations, such as memory allocation or de-allocation are performed.

4.2.4 Execution of Instrumented Programs

A run of an instrumented program P' that is not interrupted via a runtime error (i.e., by reaching *fault* configuration) reports memory leaks at the end of execution. A program run in which T_α is empty does not leak any memory. Otherwise, each element of T_α (a, b, l_a, l_u) is reported as the memory leak of size $(b - a)$ allocated at program location l_a and last known to be accessible through a variable at l_u.

4.3 Application on C Programs

The technical details of the approach as presented above are at the abstract level and need to be mapped to a concrete level to be able to apply them on a realistic programming language. This section discusses the extensions required in order to apply this approach on C programs.

4.3.1 Memory Blocks

Due to the semantics of the C programming language, where memory blocks allocated on stack are automatically freed, additionally to start and end addresses of memory blocks, their allocation types (i.e., stack, heap or global) are recorded. This enables distinguishing between memory that de-allocated dynamically (i.e., calls to functions that de-allocate memory, such as `free`) or statically (i.e., by a compiler).

4.3.2 Labels

Unlike those of the abstract language, C statements are not labelled. To generate the required information, a C program is instrumented with a stack of program locations that keeps track of entered functions and associated program locations. At any given moment of execution, the top element of the stack holds the location of the executed function, line and file, while other elements indicate locations of entered functions that lead to it.

4.3.3 Memory Tracking

At the concrete level T_α (which keeps track of the memory state of the program) is represented as look-up table, such that each element of T_α holds information about an allocated memory block and records a block's start and end addresses, allocation type and allocation and usage labels. While a lookup table can be implemented using various abstract data types, its choice should be dictated by the search operation, as it is executed most frequently (by **updateLabel**). Thus, it is imperative to be able to search quickly through the ranges of integers that represent start and end addresses of memory blocks, in order to identify whether a particular address (e.g., retrieved using the & operator) belongs to a memory block stored in T_α. This is because C supports interior pointers that do not necessarily point to the start address of a memory block. The present approach implements T_α as a red-black binary tree that uses memory addresses as keys (as memory addresses are unique integers). Due to the structure of tracked memory blocks, represented as pairs of integers, where the first element of a pair is always greater than the second, it is possible to search for an element using a particular address by testing keys against address ranges stored in nodes, proceeding to the left if the address is less than the stored start address, or to the right if it

is less than the end address. Such a search strategy guarantees that an element of T_α is found using at most $\log_2(n)$ operations. An additional argument for choosing red-black trees over, for example, AVL trees or sophisticated hash tables, is that red-black trees incur `sizeof(void*)` memory overhead per node, while other data types require more space. The rest of the operations (e.g., insert or delete) are standard.

4.3.4 Memory Allocation and De-allocation

To be able to track all allocated heap memory, alternative definitions of memory allocation and de-allocation functions are provided. In addition to their usual functionality such functions insert or remove elements of the memory tracking structure T_α. The present implementation relies on the feature of the GNU C library, where `malloc` and similar functions are implemented as weak aliases. Thus, the original definitions of functions such as `malloc` are replaced with user-defined ones. Such an approach allows all heap memory to be recorded, including blocks allocated by library functions for which no source code is available, such as `strdup`. Note that while this is adequate for experimentation, such an approach may fail to record heap memory in all cases (e.g., if memory is allocated using kernel-level functions, such as `mmap2`). In production systems, a more sophisticated approach, which modifies allocation and de-allocation functions at the kernel level, can be used.

Stack memory blocks are recorded to T_α explicitly, via inserting calls to **record** immediately after definitions of local variables. The sizes of stack blocks are determined via the `sizeof` operator. Global variables are detected statically and the call to **record** is inserted before each *use*. Stack memory blocks are removed from T_α when the scopes of their definitions are reached.

4.3.5 Memory Leak Reporting

The memory leak reporting function **report** is scheduled for execution via a call to `atexit`, which executes it before a program's termination. **report** iterates over the elements of T_α and reports heap memory in T_α (i.e., blocks that have not been de-allocated). Note that **report** is executed before the actual end of a program's scope, which makes it possible to differentiate between still reachable blocks (e.g. through global variables) and lost memory.

4.4 Empirical Evaluation

The present approach has been implemented in a prototype tool for C programs, called Skiff. Skiff is built on top of the *Clang* [144] compiler architecture (LLVM project [77]). This section reports the results of experimentation with the prototype implementation of the present approach to memory leak detection.

4.4.1 Objectives

This evaluation focuses on the value of extended memory leak reports using the **full** mode of Skiff, and on performance overheads, rather than on the number of discovered defects. This is because both techniques are sound and do not report false alarms. The reliability of Valgrind has been established by various experiments over the years. The output from Skiff has been checked to be consistent to that of Valgrind manually.

To evaluate the efficiency of the present approach a number of experiments were performed. These experiments involved instrumentation and dynamic analysis of well-known UNIX utilities, such as `find`, `grep`, `gzip`, `diff`, `patch`, `rcs`, `locate` and `rm`, and computationally intensive programs selected from the SPEC CPU [40] datasets. The main aim during experimentation was to determine the amount of runtime and memory overhead that the present approach incurs and how it compares to the existing techniques. This section also reports the results produced by Valgrind [10] (a state-of-the-art system for debugging and profiling programs) on the same test subjects and compares them to the results collected using the present approach.

4.4.2 Experiment Setup

This experimentation involved series of runs of both instrumented and original programs, and calculated overheads relative to the execution time of the original programs. To account for variance due to external factors, such as the test automation process or system I/O, the overheads are calculated using the mean over 100 runs of the modified and the original executables. A single measurement accounts for a run of a test suite (for UNIX utilities) of a single run of a program (for programs selected from SPEC).

During the experimentation with UNIX utilities the execution of test suites associated with the programs was monitored and the overheads were calculated per test suite execution. Runs of programs selected from SPEC CPU sets were performed using the test data set provided by SPEC.

The runs of Valgrind for overhead calculation were performed in a similar fashion and using the same input data.

The platform for all results reported here was an Intel Core i5-2400 3.1 GHz machine with 4GB of RAM, running Gentoo Linux.

The following section reports the results of the experimentation. The section first outlines differences in reporting and points out the benefits of locating sources of memory leaks. It then compares and discuss performance overheads incurred by different modes of Skiff and Valgrind.

```
==20077== 307,200 bytes in 1 blocks are definitely lost
==20077==    at 0x402B7B8: malloc (vg_replace_malloc.c:270)
==20077==    by 0x804A83D: loadimage (scanner.c:715)
==20077==    by 0x80489C8: main (scanner.c:1153)
```

Figure 4.6: `art`: Valgrind Report

4.4.3 Memory Leak Reports

Figures 4.6 and 4.7 show memory leak reports generated by Valgrind and Skiff in the **full** mode. This is related to the memory leak in the `art` program from the SPEC CPU2000 dataset. It can be seen that both tools report the same allocation site of the leaked block using stack traces. Note Valgrind's stack traces include function calls that occur in libraries, while Skiff's traces are limited to instrumented source code. The **full** mode of Skiff also reports the source of leaked memory shown in Figure 4.7 as `Leak source`. Skiff uses the available source code information to report details of the leaks, including names of variables (e.g., `superbuffer` in Figure 4.7) that referenced the leaked memory block prior to leakage. This removes ambiguity, as a single line of code in C may contain multiple statements. This feature may be very helpful, as C programmers take advantage of macro definitions, which often expand into complex statements spanning across one line. Further, a single location may not be sufficient to fix a defect, as the same memory block may be accessible via different variables at runtime. Skiff addresses this issue by producing a trace of usage locations prior to leakage. This feature is illustrated using the Skiff report for the CPU2000 `twolf` program[1] (see Figure 4.8), where in addition to the source of leakage Skiff reports variables that referenced a memory block prior to leakage. This is shown as part of the `Access stack`) in Figure 4.8. This information can aid in debugging as it means that developers do not need to trace the memory blocks manually. The **partial** mode of Skiff is aimed at reducing overheads, while still tracking usage locations. Thus, for carefully chosen sizes of memory blocks omitted from traversal, reports of **partial** and **full** modes match.

Figures 4.9 and 4.10 demonstrate the differences in reporting schemes between Valgrind and the **minimal** mode of Skiff. These reports uses a memory leak found in `GNU locate` (Findutils 4.4.2). Note, that apart from the differences in formatting, the information produced by both tools is equivalent. However, as shown in Section 4.4.4 Skiff detects this leak using considerably less memory and runtime overhead than Valgrind.

4.4.4 Performance Overheads

This section discusses the performance overheads of Skiff and Valgrind.

Figures 4.11 and 4.12 show the differences in memory and runtime overheads produced by a Valgrind run and a Skiff run in the **minimal** mode. The reports produced

[1] This memory leak is also used by Clause and Orso [12] to illustrate their approach.

```
* 307200 bytes
Allocation site: scanner.c:715
   [59]: loadimage [scanner.c:1153]
   [1]:  main [scanner.c:1049]
****************************************************************
Leak source: scanner.c:715 [ last known alias: 'superbuffer' ]
   [59]: loadimage [scanner.c:1153]
   [1]:  main [scanner.c:1049]
```

Figure 4.7: `art`: Skiff Report (**Full** mode)

```
Allocation site: okmalloc.c:28
   [3343]:  safe_malloc [hash.c:50]
   [3315]:  addhash     [parser.c:210]
   [132]:   parser      [readcell.c:34]
   [131]:   readcell    [main.c:71]
   [1]:     main        [main.c:22]
****************************************************************
Leak source: hash.c:25 [ last known alias: 'zapptr' ]
   [10069]: delHtab  [readnets.c:87]
   [10039]: readnets [main.c:79]
   [1]:     main     [main.c:22]
****************************************************************
Access stack:
   Variable: 'zapptr'  at hash.c:22
   Variable: 'hptr'    at hash.c:20
   Variable: 'hashtab' at hash.c:47
   Variable: 'hptr'    at hash.c:51
```

Figure 4.8: `twolf`: Skiff Report (**Full** mode)

```
==11936== 128 bytes in 1 blocks are definitely lost
==11936==    at 0x402B7B8: malloc (vg_replace_malloc.c:270)
==11936==    by 0x80515F9: xmalloc (xmalloc.c:49)
==11936==    by 0x804AAE0: search_one_database (locate.c:1106)
==11936==    by 0x804BDC3: dolocate (locate.c:1884)
==11936==    by 0x804BF43: main (locate.c:1940)
```

Figure 4.9: `locate`: Valgrind Report

```
* 128 bytes
Allocation site: xmalloc.c:49
   [35]: xmalloc [locate.c:1106]
   [33]: search_one_database [locate.c:1884]
   [8]:  dolocate [locate.c:1940]
****************************************************************
Leak source: locate.c:879 [ last known alias: 'procdata' ]
   [349]: visit_count [locate.c:375]
   [191]: visit [locate.c:385]
   [33]:  search_one_database [locate.c:1884]
   [8]:   dolocate [locate.c:1940]
```

Figure 4.10: `locate`: Skiff Report (**Minimal** mode)

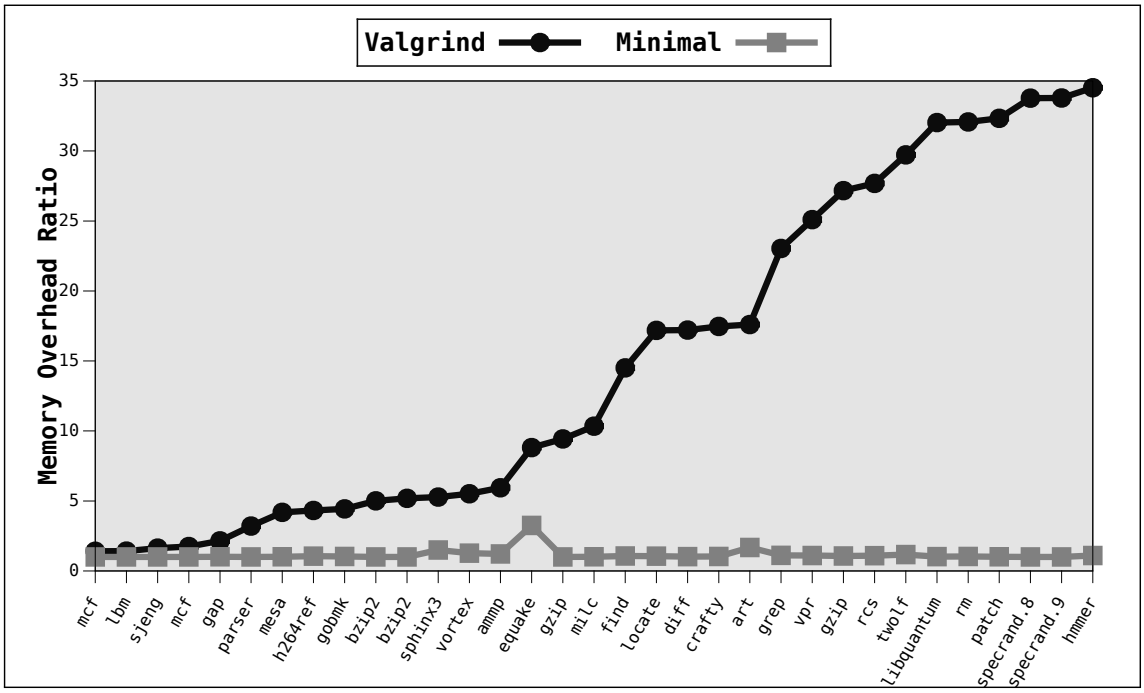

Figure 4.11: Valgrind vs. **Minimal** Mode. Memory Overhead

by both tools are similar and include detected memory leaks and their allocation sites as stack traces. Additionally, both tools report reachability of leaked memory blocks. The Y-axis measures overhead ratio (compared to the runtime or memory consumption of unmodified programs) and each point on the X-axis stands for a series of runs of a program.

It can be seen that the runtime and memory overheads produced by Skiff are lower than those of Valgrind. The memory overhead produced by Skiff averages to 1.15 times compared to unobserved execution with the highest spike of approximately 3 times in `equake`. Memory overheads of Valgrind range from 1.6 to 34 times with the average of approximately 15 times over both sets of test subjects. The runtime overheads exhibited by both tools compare similarly. The overheads produced by Skiff are on average approximately 1.8 times compared to unobserved execution, while the average runtime overhead of Valgrind is 30.8 times, ranging from 6.8 to 116 times. Note that high spikes (e.g., 116 times in `grep` or 70 times in `patch`) can be partially attributed to a high number of invocations of programs during test suite execution (e.g., over 1250 runs in `grep` test suite), where each invocation causes Valgrind to dynamically instrument a program with structures used in memory monitoring. This is different from the present approach that uses static instrumentation. Another reason for Skiff to compare favourably to Valgrind is that it tracks memory at the block level and store only the delta of information, including block addresses, sizes, locations etc., whereas Valgrind monitors each byte individually. That is, Skiff's overheads are proportional to the number of memory blocks allocated by a program, whereas Valgrind's are proportional to the overall amount of memory allocated by the program.

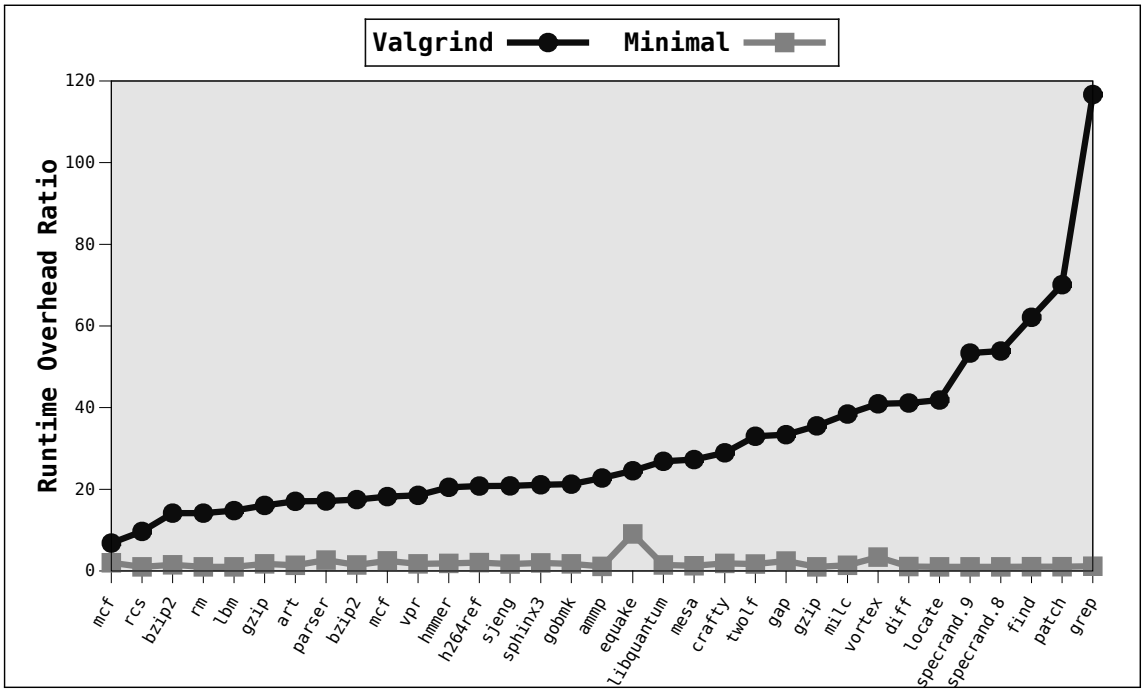

Figure 4.12: Valgrind vs. **Minimal** Mode. Runtime Overhead

Note that while results indicate that on average block-level tracking yields low overheads, the overheads are likely to increase if many small blocks are allocated. This is demonstrated via the overheads of `equake` (3 and 9 times for memory and runtime respectively). Finally, it should be noted that the **minimal** mode of Skiff monitors only allocation and de-allocation operations, which is not computationally expensive. Of course, this does not produce useful debugging information.

The runtime overheads in the **full** mode varies and increases based on the sizes of memory blocks in a program run. This is because Skiff's main runtime overhead is due to computation; that is iteration through address ranges of memory blocks and identifying pointers in assignments. Thus, the main factor that influencing Skiff's overheads is the size of the memory blocks traversed and the frequency of their use (i.e., the number of statements that trigger `updateLabel`). Consequently, larger overheads for programs selected from SPEC datasets can be expected. This is because these programs are crafted to routinely perform computationally intensive tasks (such as archiving, compilation) on large data chunks. The results of experimentation with the **full** mode of Skiff are now discussed.

Figure 4.13 compares the runtime overheads of Skiff ran in the **full** and **minimal** modes with those of Valgrind on the set of UNIX utilities. It can be seen that in the **full** mode, Skiff's overheads increase, ranging from 1.3 times compared to unobserved execution (in `gzip`) to almost 11 times in the `rcs` test suite. These overheads, however, are still lower than the overheads produced by Valgrind. Notably, the memory overhead does not increase significantly, reaching a maximum of 1.21 times in UNIX programs. The increased overheads account for extra stack memory used in recursive invocations of `updateLabel`. Figure 4.14 illustrates the increase in memory

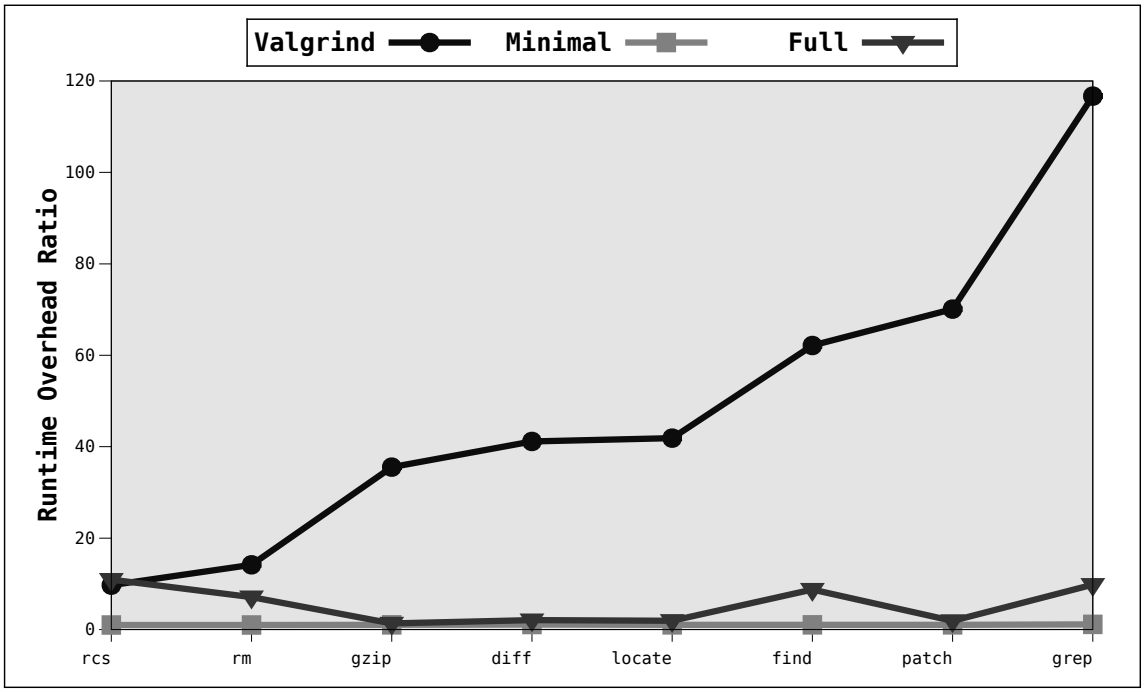

Figure 4.13: UNIX Programs Runtime Overhead

overheads in the set of UNIX utilities between the **minimal** and **full** modes (overheads incurred by Valgrind are not shown). From Figure 4.11 it can be seen that the overheads of Valgrind are approximately 15-30 times. Including the Valgrind's results in one graph would obscure the differences in performance between the two modes.

It is important to note that Skiff in the **full** mode does not always outperform Valgrind – especially for programs chosen from the SPEC CPU datasets. In some cases (e.g., `ammp`, `gzip`) Skiff's runtime overheads are extremely high (over 1000 times). The main factor that contributes to such overheads is the size of the allocated memory blocks. The larger the size of the memory allocated, the larger the overheads. This behaviour is confirmed via experimentation in **partial** mode. When traversal of memory blocks is limited by the size of the largest data structure (assuming that larger blocks are *data only* blocks and do not contain any pointers) the overheads are reduced (see Figure 4.15). For the sake of clarity Figure 4.15 does not show the data associated with Valgrind. One can compare the performances of Valgrind and Skiff in the **full** mode by combining the data from Figures 4.12 and 4.15. For example, the overheads for the `lbm` program are reduced from 566 to only 5 times compared to unobserved execution. The excessive overheads of these programs, which continue to incur large overheads in the **full** mode, is due to the structure of some SPEC programs, where a large amount of memory is allocated statically, regardless of the input size. Such an allocation pattern is rarely used in production software.

Relations between the increased runtime overheads and the amount of memory allocated by programs are illustrated in Figures 4.16, 4.17 and 4.18. In the set of UNIX utilities (see Figure 4.16) the main purpose of the associated test suites is to evaluate functional correctness of programs; thus the memory consumption does not

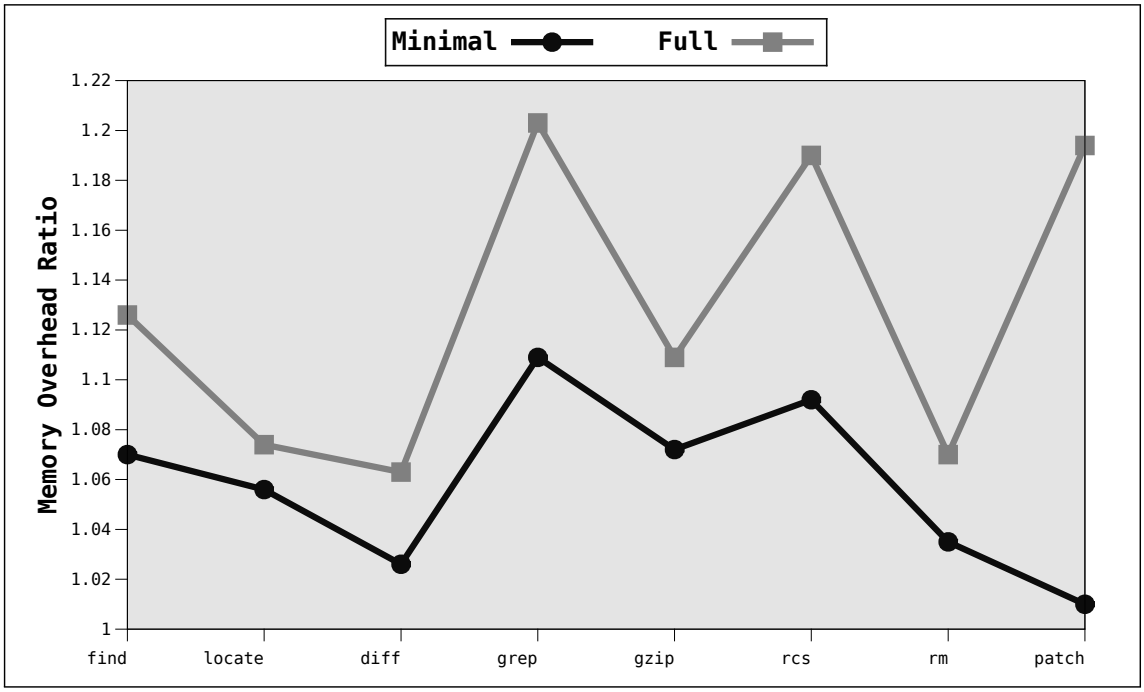

Figure 4.14: UNIX Programs Memory Overhead

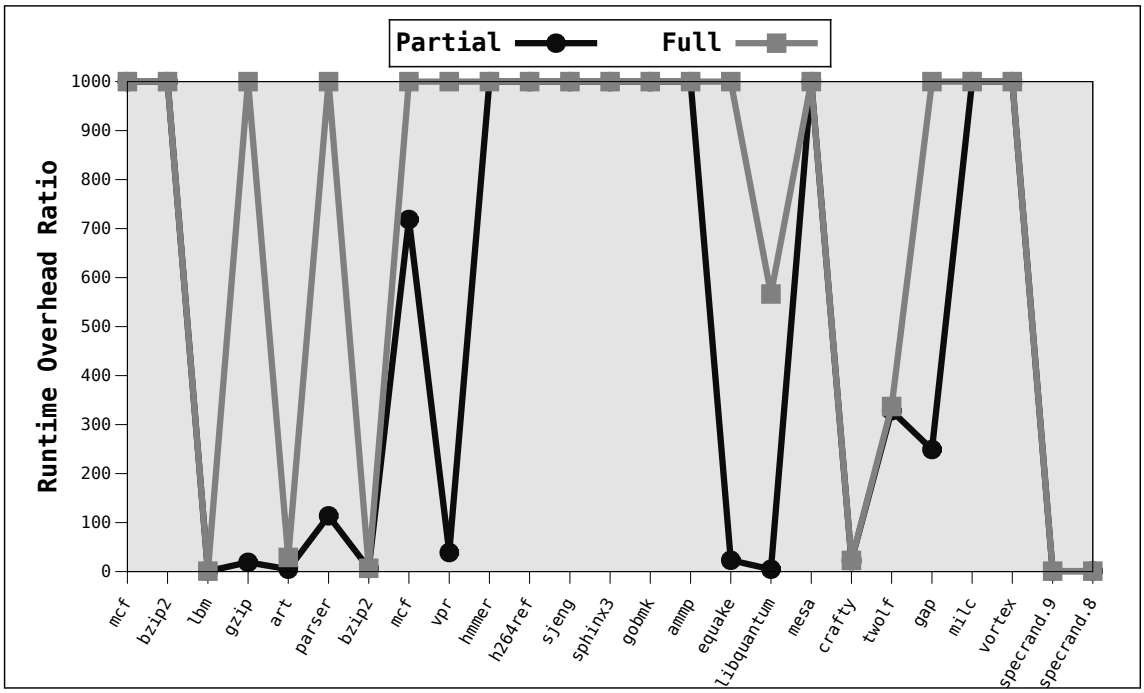

Figure 4.15: SPEC CPU Runtime Overhead

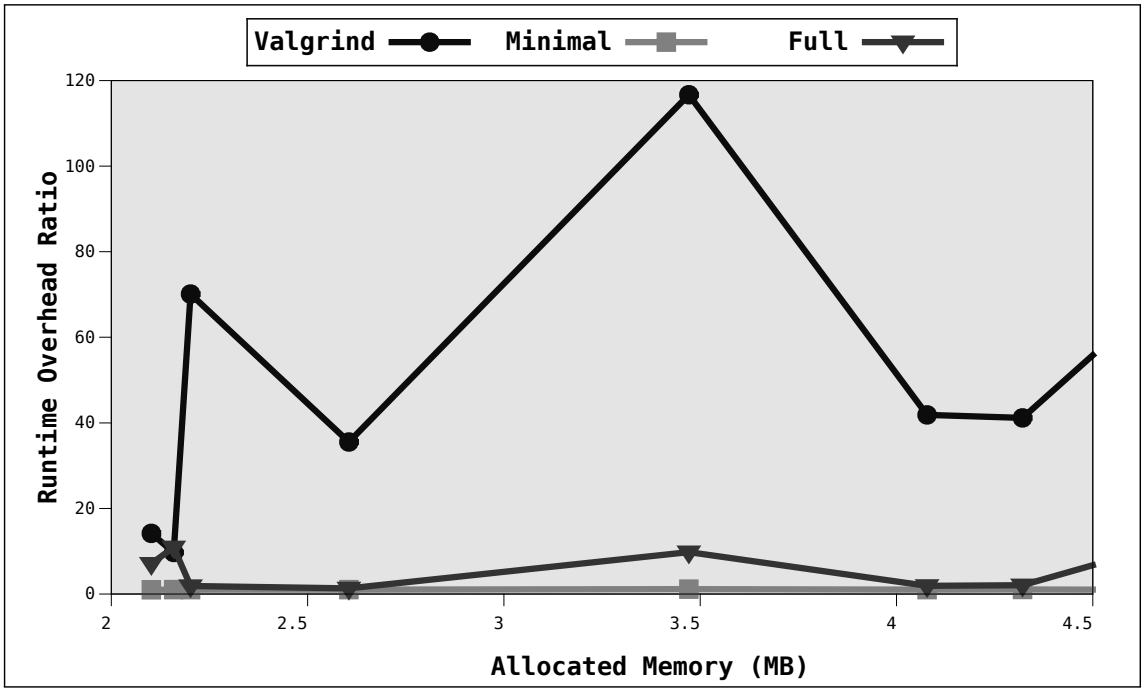

Figure 4.16: UNIX Programs Overhead Relative to Memory Usage

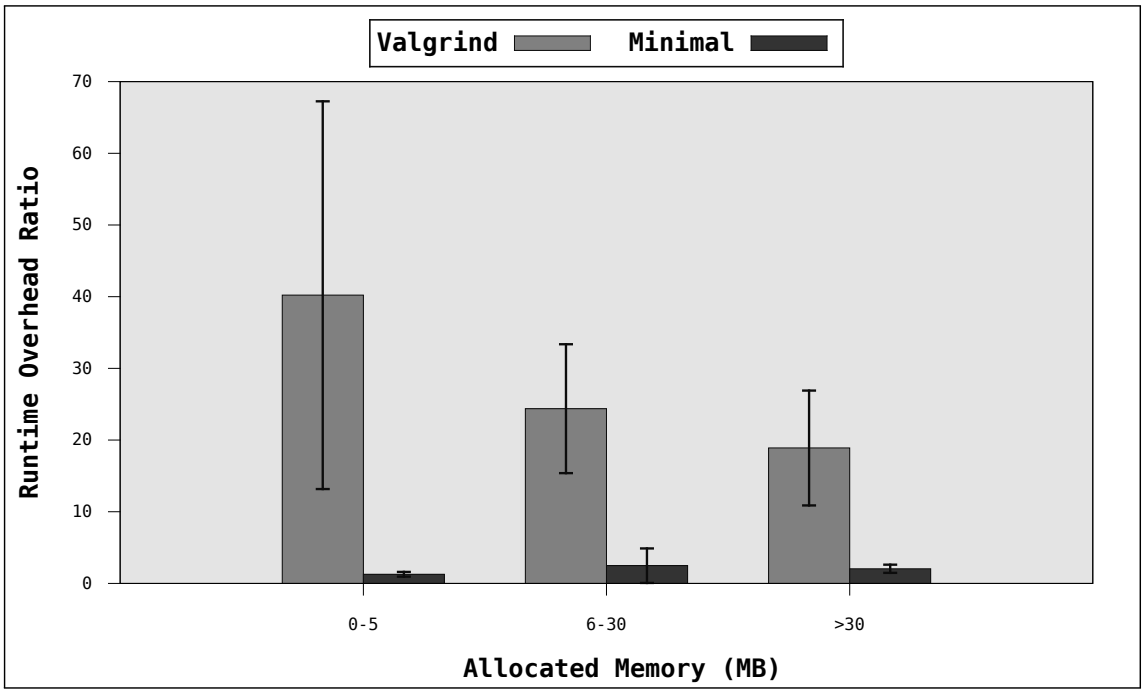

Figure 4.17: SPEC CPU Overhead Relative to Memory Usage

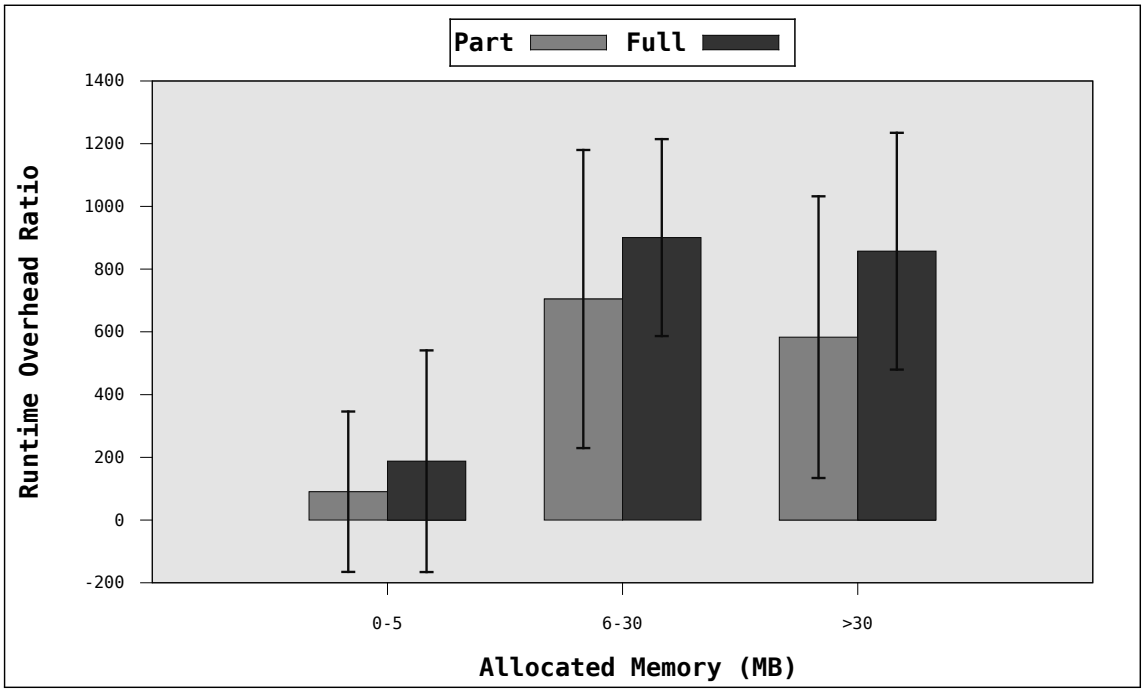

Figure 4.18: SPEC CPU Overhead Relative to Memory Usage

exceed 4.5 megabytes. Even in the **full** mode, Skiff outperforms Valgrind. In the programs selected from the SPEC CPU datasets (see Figures 4.17 and 4.18) memory consumption is much higher. It can be seen that for programs with low memory consumption (which is typical during the development process), the performance of Skiff prevails over that of Valgrind, but degrades as memory consumption grows. The presentation of these data is split because of the differences in the runtime overhead ratios. As shown in Figure 4.17, Skiff in the **minimal** mode always outperforms Valgrind. Figure 4.18 compares the overheads incurred by Skiff in the **partial** and **full** modes. Note that memory size alone does not affect the overheads. If each allocated block contains pointers, these pointers need to be tracked, adding to overheads, while if the allocated blocks contain only data, they need not be tracked, reducing the overheads. This is shown by the variance in the overheads; importantly the overheads are not consistently high.

It is noted that abnormal cases with extreme overhead in SPEC CPU should be attributed to the design pattern of SPEC programs which are aimed at performance evaluation. The memory consumption, however, affect only the **full** and **partial** modes. In the **minimal** mode the runtime overheads incurred by Skiff are negligible (i.e., the ratio of Skiff instrumentation to uninstrumented code is close to 1).

The present approach implemented in Skiff is mainly useful in the domain of functional testing, where program correctness is established through runs with small inputs. In this scenario, in addition to memory leak detection, the present technique can provide useful information that facilitates debugging. Note that the experimentation with UNIX utilities suggests that with small inputs the overheads of this approach are lower than those of conventional monitoring using Valgrind. The author's

approach can also be used in performance testing, where program runs are costly in both memory consumption and runtime. Experimentation suggests that for memory leak detection the present technique uses considerably less resources than DBI, while still producing the same level of output, especially in the **minimal** or **partial** modes.

4.4.5 Threats to Validity

This section discusses factors that may have affected the validity of the results of experimentation.

The first factor is the choice of programs and the input data used in experimentation. Even though, experimentation with UNIX utilities used realistic programs and representative inputs (i.e., test suites that are associated with the utilities), which should account for exercising most of the paths, there is no evidence that applying the present technique on different programs or using different input values will yield similar results. Similarly, during the experiment with programs selected from SPEC datasets, the input values provided may not be representative for the development process. This is because SPEC concentrates on performance evaluation, rather than on exploring various behaviours.

The second issue refers to the comparison with Valgrind. Valgrind is a memory debugger whose core functionality goes far beyond memory leak detection. Consequently, some overheads produced by Valgrind may be attributed to performing tasks that are not relevant to memory leak detection. However, Skiff is only a proof-of-concept implementation, while Valgrind is more robust. Thus, a better implementation of the present technique may improve the results.

Finally, as the overheads of monitored execution depend on the size of a pointer, the results may be affected by the architecture of the operating system. For example in a 64-bit system, where size of a pointer is twice that the size of a pointer on a 32-bit machine, slightly higher overheads may be expected.

4.5 Detecting Illegal Memory Modifications

The previous sections instantiated a technique for monitoring memory related defects and described an approach to the dynamic detection of memory leaks and leakage locations. This approach tracks elements of the memory state of a running program, including memory allocations and associated program locations (i.e., allocation and usage labels). The nature of the tracked information (e.g., boundaries of allocated blocks) suggests that this technique can be adapted for the detection of different memory-related related defects, such as illegal memory dereferencing or free operations. For example, to detect illegal free errors, it is sufficient to monitor invocations of memory de-allocation function `free` and check whether its inputs correspond to the start addresses of allocated memory blocks. Similarly, to detect illegal dereferences

errors, one can check whether dereferenced addresses lie within the boundaries of blocks belonging to the memory allocation.

While such extensions are straightforward, it is not clear whether the performance overheads of such an approach will continue to compare favourably to the results of state-of-the-art techniques, such as Valgrind. This section presents an extension aimed at detecting illegal memory modifications (i.e., modifications of memory locations outside of the memory allocation). It first describes the details of this extension at the level of the abstract imperative language (see Figure 4.1). This section then discusses how to adapt this extension for monitoring of C programs. Finally, the section presents the results of preliminary evaluation using computationally expensive programs selected from the SPEC CPU datasets, comparing the results of the prototype implementation to the results of Valgrind (which implements similar checks). This comparison shows that the present approach results in lower runtime overheads than those of Valgrind.

4.5.1 Extension at the Abstract Level

This section describe the extension for detecting illegal memory modifications at the level of the abstract imperative language (see Figure 4.1). Its syntax, operational semantics and memory model are discussed in Section 4.1.

Memory Semantics

The Operational semantics of illegal memory modifications is given via Rule $VarAsgn_2$ (see Figure 4.3), leading to a runtime error via the special configuration *fault*. That is, an illegal modification of memory occurs via a memory assignment $\langle l : \mathbf{deref}(v) := e \rangle$, if the dereferenced address (given by the value variable v evaluates to) does not belong to the memory allocation α.

Monitoring Commands

In order to detect memory modifications at the level of the abstract language the set of monitoring commands $Comm_m$ is extended with the command $\mathbf{checkDereference}(T_\alpha, a, l)$ to detect and report locations of such errors. Given a memory tracking state $T_\alpha \in Mt$, address $a \in \mathbb{N}$ and a label $l \in Lab$, **checkDereference** performs a lookup in T_α and returns 1 if the address a lies within one of the memory blocks tracked by T_α (which indicates that the dereference is valid), or 0 otherwise. That is,

$$\mathbf{checkDereference}(T_\alpha, a, l) = \begin{cases} 1 & \text{if } \exists (c, d, l_a, l_u) \in T_\alpha : c \leq a \leq d \\ 0 & \text{otherwise} \end{cases}$$

$$\textit{CheckDereference:}\ \frac{}{\langle \textbf{checkDereference}(T_\alpha, a, l), T_\alpha, m, \sigma\rangle \rightarrowtail_m \langle \texttt{skip}, T_\alpha{}', m, \sigma\rangle}$$

Figure 4.19: Operational Semantics of a **checkDereference**

$$\textit{VarAsgn:}\ \frac{}{\langle l : v := e\rangle \quad \rightsquigarrow \quad \langle l : v := e\rangle}$$

$$\textit{MemAsgn:}\ \frac{}{\langle l : \textbf{deref}(v) := e\rangle \quad \rightsquigarrow \quad \begin{array}{l}\textbf{checkDereference}(T_\alpha, v, \textbf{loc});\\ \langle l : \textbf{deref}(v) := e\rangle;\end{array}}$$

Figure 4.20: Syntactic Transformations for Illegal Dereference Detection

Additionally, if a call to **checkDereference**(T_α, a, l) returns 0 (i.e., an illegal memory modification error is detected), this function reports the location of the occurred error using label l.

Formally, the operational semantics of **checkDereference** is shown in Figure 4.19. Since this function only detects and reports invalid memory modifications, it does not result in a change in the memory tracking state, memory allocation or memory mapping.

Syntactic Transformations

The syntactic transformations that enable checking for illegal memory modifications are shown in Figure 4.20. The Rule *MemAsgn* is used to monitor memory assignments for illegal memory modifications using the **checkDereference** function. Every command $\langle l : \textbf{deref}(v) := e\rangle$ that assigns a value (given by the result of evaluation of expression e) to an address in the memory (given by the value bound to variable v) is preceded by the call to **checkDereference**. This reports an error at location l if the address given by the value bound to the variable v does not belong to the memory allocation and therefore leads to an illegal memory modification. Note that since illegal memory modifications occur only through memory assignments, no transformations are performed for variable assignments (see Rule *VarAsgn* in Figure 4.20).

The rest of the transformations required to capture illegal memory modifications are shown in Figure 4.5. Note that memory allocation tracking relies on calls **insert** and **delete**, which update the memory tracking state T_α as the modified program executes.

4.5.2 Application on C Programs

This section discusses issues related to capturing illegal memory modifications at the level of the C programming language.

At the concrete level of C the monitoring function `checkDereference` is used to check program dereferences via `*`, `->` and `.` operators and array subscripts. This captures all illegal memory modifications (as a memory needs to be dereferenced before it is written to) and additionally identifies whether accesses to memory made by a program run are valid. That is, before an address a is dereferenced by a run of a program, it is checked that a exists in the memory allocation. If a is determined to lie outside of any of the allocated blocks, an illegal dereference is reported. It is noted, that this concrete level of monitoring, does not differentiate between memory accesses or modifications and reports errors as illegal dereferences.

The implementation of `checkDereference` straightforward. This function receives a memory address a as input, checks whether a belongs to the memory allocation by performing a lookup in the global memory tracking structure implemented as a red-black binary tree and reports an illegal dereference error at a currently executed source location if a does not belong to a program's memory allocation.

Since illegal memory modifications can occur in expressions in C (e.g. `*p++`, where `p` is a variable), dereference checking is enabled by rewriting dereference expressions to statement expressions that check the validity of dereferences before they occur. For example, a dereference via operator `*` (say `*p`, where `p` is a variable), is rewritten to expression `*({checkDereference(p); p;})`. This first evaluates the validity of the dereference of the pointer variable `p` and then evaluates to the value of the memory given by the address `p` points to. Array subscripts are rewritten similarly. For example, expression `a[i]` (where `a` is an array, say `char a[10]`, and `i` is an integer, say `int i`), rewrites to `*({checkDereference(a+i); (a+i);})`. This is because in C arrays and pointers are handled similarly. Dereferences via `.` and `->` operators on structs and unions are checked using the `offsetof` operator, which evaluates to the byte offset of a given member within a struct or union. That is, structs and unions can be handled similarly to arrays, where an index is given by the application of `offsetof`. For example, expression `a->b`, where `a` is a struct that has a member `b`, is rewritten to `a->({checkDereference(a + offsetof(a,b)); b;})`.

The remaining elements of the technique at the concrete level of C are described in Section 4.3.

4.5.3 Experimentation Results

This section presents the results of experimentation with the prototype implementation of the technique for detecting illegal memory modifications and accesses.

To evaluate the overheads the present approach memory modifications and accesses were checked in the runs of computationally intensive programs selected from SPEC CPU datasets. Programs selected from SPEC, inputs and experimental setup is consistent with the experimentation with memory leaks described in Section 4.4.

The main aim of the experimentation was to determine the amount of runtime

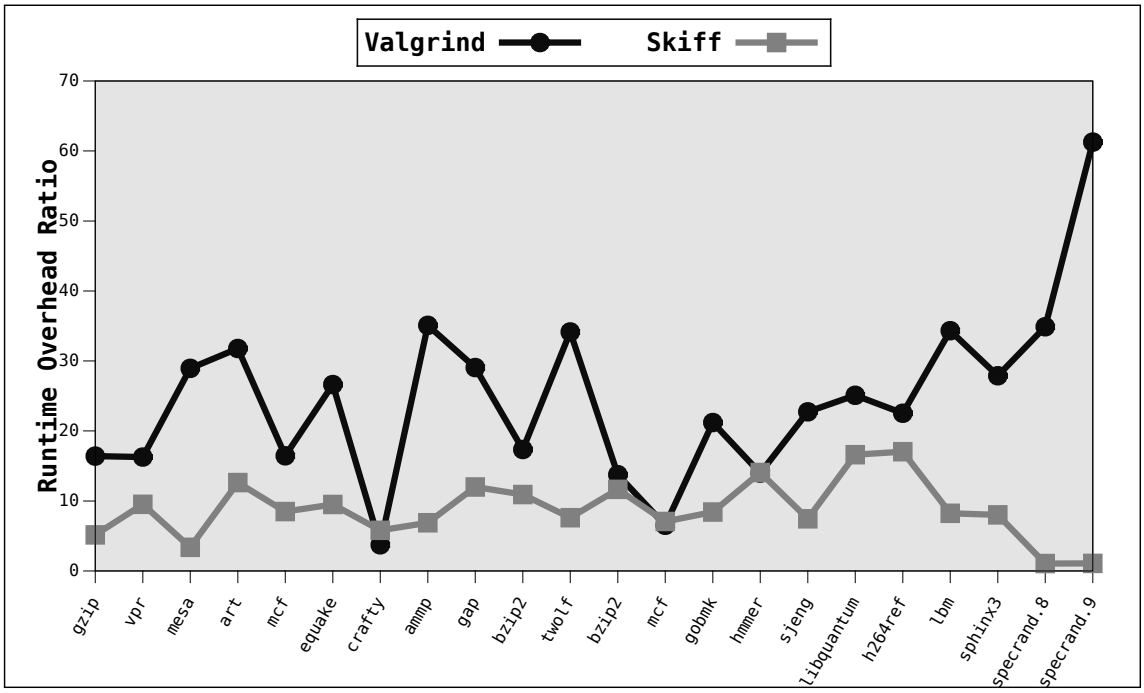

Figure 4.21: Valgrind vs. Skiff. Runtime Overhead

overhead this extension would produce and how it compares Valgrind that implements similar checks. In other words, the goal is identify whether the present approach applied to a different problem continues to result in lower overheads than those of Valgrind. Memory overheads are not investigated; The memory overheads of Valgrind were discussed in Section 4.4. Memory overheads of the prototype are similar to the overheads for memory leak detection. This is because same information as for memory leak detection in the **minimal** mode is tracked.

This section report the results of Skiff extended to detect illegal memory modifications and accesses using computationally intensive programs from the SPEC datasets, and compares them to those of Valgrind using the same set of test subjects. It is noted that the results presented in this section are preliminary and have not appeared in published work.

The results of the experimentation are shown in Figure 4.21. The results indicate that the present approach applied to a different problem continues to result in considerably lower runtime overheads than Valgrind's. The runtime overheads of Skiff range from 1.05 times in `specrand` to approximately 17 times compared to unobserved execution in `h264ref` program from SPEC CPU 2006 dataset. On average, for the programs from SPEC datasets the Skiff overheads are approximately 8.6 times the unobserved execution. The overheads of Valgrind are higher, averaging approximately 24.5 times, with a minimum of 3.7 times and a maximum of 61 times. Note that same as in the experimentation with memory leaks, Skiff does not always outperform Valgrind. For example, on the `crafty` program from SPEC CPU 2000 the overheads of Skiff are approximately 5.8 times compared to unobserved execution, whereas Valgrind incurs the overheads of approximately 3.7 times.

The Skiff's overheads can be attributed to the number of invocations of **checkDereference** executed for every dereference, array application and struct access. That is, Skiff's overheads are proportional to the number of dereferences in a run of a program. Another factor that contributes to the overheads of Skiff is the number of allocated and de-allocated memory blocks (which includes variable definitions). This is because to detect illegal dereferences Skiff needs to track memory allocation. That is, this extension additionally implements the **minimal** mode for memory leak detection (since each allocated block is recorded along with the location of its allocation). However, the results of experimentation with the **minimal** mode (Figure 4.12) indicate that the runtime overheads of Skiff for the case where only locations of allocations are recorded average to 1.8 times the unobserved execution. Therefore, for this analysis, the number of invocations of **checkDereference** (i.e., number of lookups performed in the data structure that captures memory allocation) is the main contributing factor for the incurred overhead.

Threats to Validity

Threats to the validity of the results of this experimentation are similar to those factors discussed in Section 4.4.5, and include the choice of the programs and the input data. Another important factor, is that in addition to memory leak detection and dereference checking Valgrind implements checks for uninitialised values. However, Skiff checks the validity of dereferences on the stack and on the heap, while Valgrind checks read and write accesses for heap blocks only. Finally, a different implementation of `checkDereference` may yield different results. For example, the current implementation uses a red-black tree to store memory allocation. An approach that uses shadow memory is likely to result in lower overheads for lookups, but also to require more memory to track allocation.

4.6 Concluding Remarks

This chapter presented a tunable monitoring technique for detecting memory leaks and locations of leakage. The technique uses source-to-source transformations to instrument an input program with statements to monitor its memory state and report leaks before the modified program terminates. One of the main benefits of this approach is the ability to locate the sources of where memory was lost. Additionally, the proposed approach provides tuned monitoring via different modes of execution enabled at runtime. In **full** mode extra information of leakage locations is produced for the cost of larger overheads. **Minimal** mode reduces overheads using a conventional reporting scheme that outputs only allocation sites, while **partial** mode reduces overheads by tracking the leakage locations of memory blocks of specified sizes only.

This chapter also showed that the monitoring primitives required to detect memory leaks and associated leakage locations are sufficient to enable detection of other types of issues. This has been demonstrated by an extension designed to detect illegal modifications of memory.

The technique discussed in this chapter has been implemented in a research prototype for monitoring of C programs called Skiff. The approach was evaluated using experimentation that monitored real UNIX utilities and computationally expensive programs from SPEC CPU datasets, and compared the results of Skiff with those of the memory debugger Valgrind. The results show that for the problem of memory leak detection (where only locations of allocations of leaked memory were tracked) Skiff outperforms Valgrind. This suggests that Skiff may be used as a replacement for binary instrumentation tools, producing similar results with considerably fewer system resources.

Experimentation with **full** mode (which enables detection of locations of leakage) show that the overheads of Skiff directly depend on the amount of memory allocated by programs, and they increase as memory consumption grows. For monitoring of UNIX utilities Skiff performed better than Valgrind mainly due to relatively small allocated blocks. However, Skiff performed considerably worse on programs selected from the SPEC datasets, which focus on performance evaluation and thus use large inputs. Further, the experimentation demonstrated the applicability of overhead tuning using **partial** mode, where in some cases large overheads of SPEC programs were reduced by not tracking large data blocks for leakage. At present, it can be suggested that for the detection of leakage, the proposed approach is mainly useful in the domain of functional testing, where correctness is determined using runs with small inputs.

This chapter also presented preliminary results of evaluation with the extension that enables detection of illegal memory modifications. For this experimentation runs of computationally expensive programs selected from the C SPEC CPU datasets were used and the Skiff's results were compared to the results of Valgrind, which implements similar checks. This experimentation indicates that the runtime overheads of Skiff are considerably lower than those of Valgrind. The results for memory overheads are consistent with those for memory leak detection.

5

A Value Tracking Approach to Information Flow Security

This chapter investigates the runtime detection of issues related to leakage of confidential information used by a program. Similarly to the previous chapter, which focuses on the detection of memory leaks, the aim of this investigation is to develop precise monitoring techniques with overheads acceptable for use with testing.

This chapter describes a monitoring approach to the detection of information leakage via assignment of secret values to unsafe program locations (e.g., publicly visible variables). It is shown that this approach detects such issues as password disclosure and a number CWE [145] vulnerabilities related to handling of sensitive data. The results of the empirical evaluation with the prototype implementation for C programs indicate that the overheads for detecting of password disclosure in real software does not exceed 1%. The overheads associated with the detection of CWE vulnerabilities are still acceptable for use with testing, but incur higher overheads.

Some of the elements of the approach to information leakage and the initial results of the empirical evaluation of password flow presented in this chapter previously appeared in a conference publication [146]. An extended version of this paper, which includes the results of the experiments with CWE vulnerabilities, is currently being considered for publication in the *Information and Software Technology Journal*.

The suggested technique analyses program values and has the ability to identify whether a disclosed value represents an information leak with respect to the values considered secret at runtime. This differs from the majority of the existing techniques,

which analyse programs with respect to its variables and track security labels or propagate taint marks. Tracking only a handful of values whose disclosure constitutes information leakage reduces the overheads associated with tracking.

This approach assumes that statements that generate secret data are identified using manual annotations to the program; that is, the program locations of assignments that transfer secret values to program variables are marked. An input program P is instrumented (via a series of source-to-source transformations) with statements that track secret values and safe locations, and assertions that check the safety of assignments with respect to the tracked values. This generates a modified program P'. A run of P' observes the execution of the original program P by detecting information leakage via assignment of secret values to unsafe locations. A program run that has no detected assertion failures does not leak the secret values captured at runtime via the annotated assignments.

The proposed approach is supported by a prototype implementation for C programs that is used to conduct various experiments. The results of initial experimentation show that this approach can be used to address narrow yet practical problems, such as preventing leakage of passwords. Monitoring the safety of password flow in a number of security-oriented UNIX utilities indicates that this dynamic analysis of secret values results in low overhead of 1%, while still soundly identifying information leakage in real security software. Further experimentation demonstrates that this technique is a good fit for analysing programs for security vulnerabilities related to information leakage stressed by security-oriented communities, such as the Community Developed Dictionary of Software Weakness Types [145].

This chapter also reports on experiments using a number of computationally expensive programs from the SPEC datasets. This addresses issues related to information leaks via the de-allocated but not cleared out memory, improper handling of sensitive data (e.g., plain-text storage or hard-coding of passwords), exposure of sensitive information through standard output channels and information leaks via temporary files and file handles. The results show that the author's approach handles complex programs, such as `gcc`, while still yielding acceptable overheads. Finally, the same properties are used to analyse popular security-oriented software such as `openssh` and `ccrypt`; this analysis shows that overheads incurred by the author's approach remain low.

This chapter offers the following contributions:

- A value-tracking approach to detecting information leakage due to disclosure of secret values used by a run of a program.
- A proof-of-concept implementation of the present approach for C programs.
- An empirical evaluation that concentrates on overheads incurred by the monitored execution.

$$
\begin{array}{lcl}
v & ::= & x \mid \textbf{ptr}\, v \\
e & ::= & n \mid v \mid e \oplus e \mid f(e) \mid \textbf{addressof}(v) \\
c & ::= & \texttt{skip} \mid \textbf{def}(v) \mid v := e \mid \langle v := e \rangle \mid \texttt{assert}(e) \mid \\
 & ::= & \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid c_1\,;c_2 \\
f & ::= & \mathit{Ident} \triangleq c \mid f_1\,;f_2 \\
P & ::= & f\,;e
\end{array}
$$

Figure 5.1: Abstract Language

The remainder of this chapter is organised as follows. Section 5.1 presents extensions to the standard model, first presented in Chapter 3. These extensions capture the semantics of information leaks and are further used to describe the present approach at an abstract level in Section 5.2. Section 5.3 discusses how to apply the proposed approach on C programs and Section 5.4 presents the results of the empirical evaluation using the prototype implementation for C programs. Finally, Section 5.5 offers concluding remarks.

5.1 Syntax and Semantics

This chapter present a monitoring approach to the detection of information leakage using an abstract imperative language (Chapter 3) extended with operations on pointers, assertions and source code annotations. These extensions allow for the definition and therefore detection of the disclosure of sensitive information at runtime. This section describes the syntax, memory and operational semantics of the extensions used for information leak detection.

5.1.1 Syntax

Figure 5.1 presents an abstract imperative language extended with operations on pointers, assertions and source code annotations. The following outlines the syntactic extensions to the standard model (see Figure 3.1) that help to capture information leaks.

Variables v (given by the set *Var*) are partitioned into variables that hold primitive values (indicated by x) and variable references (**ptr** v), where **ptr** is a syntactic type annotation. The set of program expressions *Expr* is extended with the **addressof** operator on program variables. The set of program commands (*Comm*) is extended with annotated assignments ($\langle v := e \rangle$) and assertions $\texttt{assert}(e)$. The rest of the elements of the language are standard and have been discussed in detail in Chapter 3. Before discussing these extensions, it is useful to describe elements of the memory model for information leak detection.

5.1.2 Memory Semantics

The memory semantics of the extension is consistent with that of the standard model. Accordingly, values and memory addresses are given by the set of natural numbers $\mathbb{N}$. Memory mapping in a particular state is represented by the function $m : \mathbb{N} \to \mathbb{N}$, which maps memory addresses to values, and function $\rho : \mathit{Var} \to \mathbb{N}$ represents variables in the memory. Discussion on the memory semantics of the standard model is given in Section 3.2.

The following discusses extensions to the memory semantics of the standard model that allow for the representation of information leaks.

In a particular state of computation, a set of secret values (i.e., values that need to be protected against disclosure) is given by the set of values S_v, such that S_v is an element of $\mathscr{P}(\mathbb{N})$. The set of values $S_a \in \mathscr{P}(\mathbb{N})$ is used to represent the set of safe memory addresses in a particular state. A *safe* memory address represents a memory location that cannot be observed by an adversary; that is, during a run of a program, a value mapped to a safe address is never disclosed to a third party. Unsafe addresses, therefore, represent publicly observable memory locations. It is assumed that in a program run any location is publicly observable, unless it is given by a safe address. Accordingly, any value mapped to an unsafe address (i.e., that does not to S_a) can be viewed by an attacker. Further, it is said that a memory location is *safe* if it cannot be observed by a third party (i.e., it is represented by a safe address) and unsafe otherwise.

5.1.3 Operational Semantics

This section describes evaluation of expressions and the operational semantics of the commands of the imperative language used to describe the present approach (see Figure 5.1).

Evaluation of Expressions

Similar to the standard model, the evaluation of a program expression $e \in \mathit{Expr}$ is given by function **eval** : $(\mathit{Expr} \times \mathit{Mem}) \to \mathbb{N}$ where $m \in \mathit{Mem}$ is a memory mapping. The evaluation of core expressions is shown in Figure 5.2.

The functions *l-value* and *r-value* are used to distinguish between *l-values* and *r-values* of variables respectively. Given some variable $v \in \mathit{Var}$, function $\mathit{l\text{-}value} = \mathit{Var} \times \mathit{Mem} \to \mathbb{N}$ returns a memory location (i.e., an address) associated with v and function $\mathit{r\text{-}value} = \mathit{Var} \times \mathit{Mem} \to \mathbb{N}$ returns the value to which variable v evaluates in the memory. For a primitive variable x, *l-value* returns its memory address: that is, $\mathit{l\text{-}value}(x, m) = \rho(x)$, where m is a memory mapping. For a variable reference $\mathbf{ptr}\, v$, *l-value* returns the address of the memory location that v references, i.e., $\mathit{l\text{-}value}(\mathbf{ptr}\, v, m) = \mathit{r\text{-}value}(v, m)$. Evaluation of variables in the memory is given via

$l\text{-}value(x,m)$	$= \rho(x)$
$l\text{-}value(\mathbf{ptr}\,v,m)$	$= r\text{-}value(v,m)$
$r\text{-}value(v,m)$	$= m(l\text{-}value(v,m))$
$\mathbf{eval}(v,m)$	$= r\text{-}value(v,m)$
$\mathbf{eval}(\mathbf{addressof}(v),m)$	$= l\text{-}value(v,m)$

Figure 5.2: Evaluation of Program Expressions

r-value, such that $r\text{-}value(v,m) = m(l\text{-}value(v,m))$. That is, a primitive variable x evaluates to the value to which its address is mapped in the memory: $m(\rho(x))$. A variable reference $\mathbf{ptr}\,v$ evaluates to the value mapped to the address of the memory block it references: $m(m(\rho(v)))$. The evaluation of variables in a program is thus defined via *r-value*, and the evaluation of expression $\mathbf{addressof}(v)$, where **addressof** is an operator that identifies memory locations and v is a variable, is given by *l-value*.

Evaluation of numerals, binary expressions and function calls is standard (see Section 3.3.1).

Operational Semantics of Commands

The operational semantics of commands of the abstract language is shown in Figure 5.3 and defined as a relation $\rightarrowtail$ on configurations: $\langle c : m, S_a, S_v\rangle$ and *abort*, where $c \in Comm$ is a program command, $m \in Mem$ is a memory mapping, $S_a \in \mathscr{P}(\mathbb{N})$ is a set of safe addresses and $S_v \in \mathscr{P}(\mathbb{N})$ is a set of secret values. *abort* is a special configuration that leads to an abrupt program termination. Configuration $\langle \texttt{skip} : m, S_a, S_v\rangle$ is final.

The operational semantics of assignments (given via Rule *Asgn*, Figure 5.1.4) is similar to the standard model (see Section 3.3.2, Chapter 3). That is, assignments $v := e$, where v is a variable in *Var* and e is an expression in *Expr* replace the value mapped to the address of a variable v (given via $l\text{-}value(v,m)$ where m is a memory mapping) with the result of evaluation of expression e (i.e., $\mathbf{eval}(e,m)$).

Annotated assignments $\langle v := e\rangle$ (see Figure 5.1.4, Rule *Annotated*) represent the assignment of secret values to safe locations. That is, an annotated assignment $\langle v := e\rangle$ transfers a secret value (given via the expression e) to a safe location given by the address of variable v (shown via the updated sets of secret values and safe locations ${S_a}^*$ and ${S_v}^*$). Further, since construct $\langle c\rangle$ (where c is a command) is only a syntactic annotation, command $\langle v := e\rangle$ executes assignment $v := e$.

The operational semantics of assertions is given via Rules $Assert_1$ and $Assert_2$. $\texttt{assert}(e)$ (where e is an expression that evaluates to a zero value) results in an abrupt program termination (i.e., executes the special configuration *abort*, Rule $Assert_1$). For the case when e evaluates to a non-zero value (indicated via the side condition $\mathbf{eval}(e,m) \neq 0$ in Rule $Assert_2$) $\texttt{assert}(e)$ is equivalent to `skip`.

Operational semantics of the remaining commands, variable definitions $\mathbf{def}(v)$

$$\textit{Def:}\ \frac{}{\langle \mathbf{def}(v) : m, S_a, S_v\rangle \rightarrowtail \langle \texttt{skip} : m, S_a, S_v\rangle}$$

$$\textit{Asgn:}\ \frac{}{\langle v := e : m, S_a, S_v\rangle \rightarrowtail \langle \texttt{skip} : m[\![\textit{l-value}(v,m) \mapsto \mathbf{eval}(e,m)]\!], S_a, S_v\rangle}$$

$$\textit{Annotated:}\ \frac{}{\langle\langle v := e\rangle : m, S_a, S_v\rangle \rightarrowtail \langle v := e : m, {S_a}^*, {S_v}^*\rangle}$$

where:

$${S_a}^* = S_a \cup \textit{l-value}(v,m)$$
$${S_v}^* = S_v \cup \mathbf{eval}(e,m)$$

$$\textit{Assert}_1\textit{:}\ \frac{}{\langle \texttt{assert}(e) : m, S_a, S_v\rangle \rightarrowtail \textit{abort}}\ (\mathbf{eval}(e,m) = 0)$$

$$\textit{Assert}_2\textit{:}\ \frac{}{\langle \texttt{assert}(e) : m, S_a, S_v\rangle \rightarrowtail \langle \texttt{skip} : m, S_a, S_v\rangle}\ (\mathbf{eval}(e,m) \neq 0)$$

$$\textit{Seq}_1\textit{:}\ \frac{\langle c_1 : m, S_a, S_v\rangle \rightarrowtail \langle c_1' : m', {S_a}', {S_v}'\rangle}{\langle c_1\,;c_2 : m, S_a, S_v\rangle \rightarrowtail \langle c_1'\,;c_2 : m', {S_a}', {S_v}'\rangle}$$

$$\textit{Seq}_2\textit{:}\ \frac{\langle c_1 : m, S_a, S_v\rangle \rightarrowtail \langle \texttt{skip} : m', {S_a}', {S_v}'\rangle}{\langle c_1\,;c_2 : m, S_a, S_v\rangle \rightarrowtail \langle c_2 : m', {S_a}', {S_v}'\rangle}$$

$$\textit{If}_1\textit{:}\ \frac{}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : m, S_a, S_v\rangle \rightarrowtail \langle c_1 : m, S_a, S_v\rangle}\ (\text{where } \mathbf{eval}(e,m) \neq 0)$$

$$\textit{If}_2\textit{:}\ \frac{}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : m, S_a, S_v\rangle \rightarrowtail \langle c_2 : m, S_a, S_v\rangle}\ (\text{where } \mathbf{eval}(e,m) = 0)$$

$$\textit{While:}\ \frac{}{\langle \texttt{while}\ e\ \texttt{do}\ c : m, S_a, S_v\rangle \rightarrowtail \langle \texttt{if}\ e\ \texttt{then}\ (c\,;\texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip} : m, S_a, S_v\rangle}$$

Figure 5.3: Operational Semantics of Program Commands

(Rule *Def*), conditionals `if` e `then` c_1 `else` c_2 (Rules If_1 and If_2), loops `while` e `do` c (Rule *While*) and sequential compositions of commands $c_1 \ ; c_2$ (Rules Seq_1 and Seq_2) are standard and have been discussed in detail in Section 3.3.2.

5.1.4 Information Leak

The following provides a definition of an information leak.

Informally, an information leak occurs when a secret value or a value that is '*close enough*' to one of the secret values in a program run is assigned to an unsafe location. For instance, a value k that needs to be protected against disclosure, or values that are similar, but not identical to k, become publicly observable. The exact definition of '*close enough*', which denotes similar but not identical values, can be customised, but for any given implementation it is fixed. For the purpose of this abstract presentation it is assumed that this measure of similarity between values is given by function **isSimilar** : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ that takes two values and returns a non-zero value if the input values are '*close enough*' and zero otherwise.

The following provides a formal definition an information leak occurring at runtime.

Definition 10 (Information leak) *Let $S_a \in \mathscr{P}(\mathbb{N})$ be the set of values that represent addresses of safe locations, $S_v \in \mathscr{P}(\mathbb{N})$ be the set of secret values, $c, c' \in Comm$ be program commands, $m \in Mem$ is a memory mapping and $k, a \in \mathbb{N}$ be values. The following property defines an **information leak** that occurs as a result of the execution of the command c in the memory mapping m with respect to the set of addresses of safe locations S_a and the set of secret values S_v:*

$$Leak(c, m, S_a, S_v) \iff \langle c, m\rangle \rightarrowtail_m \langle c', m[\![a \mapsto k]\!]\rangle \wedge a \notin S_a \wedge \exists x \in S_v : \textbf{isSimilar}(k, x) \neq 0$$

That is, there exists a transition $\langle c, m\rangle \rightarrowtail_m \langle c', m[\![a \mapsto k]\!]\rangle$ in the program that associates a memory address given by the value a to the value k in the memory mapping $m[\![a \mapsto k]\!]$. Since assignment is the only command that modifies the memory mapping of the program, this transition is the result of the execution of some assignment $v := e$, such that the address of the variable v is given by the value a and k is the result of evaluation of expression e in the memory mapping m. That is, the execution of the assignment $v := e$ results in an **information leak** (denoted as $Leak(c, m, S_a, S_v)$) if and only if a represents an unsafe location (i.e., does not belong to the set of addresses of safe locations given by S_a) and the value k is '*close enough*' to one of the secret values from S_v. The latter is determined by the function **isSimilar**, which compares two input values and returns a non-zero value if the input values are determined to be '*close enough*'.

Using this formal definition of an information leak, the following section describe details of the present monitoring approach to information leak detection.

5.2 Information Leak Detection

The present analysis makes source-to-source transformations on an input program P containing annotated assignments that mark the assignment of secret values (i.e., values that need to be protected against disclosure) to *safe* memory locations that are not publicly observable. These transformations yield a modified program P'. A run of P' tracks the secret values and addresses of safe locations and prevents disclosure of secret values by aborting the execution if an information leak is detected.

This section describes the details of runtime information leak detection. It first describes the elements of a monitoring state (i.e., data that needs to be tracked to detect information leaks). This section then discusses the semantics of monitoring commands and presents the set of transformation rules used to derive the instrumented programs. Finally, it describes how the execution of the modified programs prevents information leaks at runtime.

5.2.1 Monitoring State

To keep track of the data required to capture information leaks during the execution of a modified program P', collections of values H_{val} and H_{var} are used. H_{var} represents a collection that tracks addresses of safe locations and H_{val} represents a collection of secret values that should be protected against disclosure. Formally, H_{var} and H_{val} are given by sets of values, that is $H_{var}, H_{val} \in \mathscr{P}(\mathbb{N})$. Further, a monitoring state of a modified program is represented by a pair of sets (H_{var}, H_{val}), where H_{var} tracks addresses of safe locations and H_{val} captures secret values.

5.2.2 Semantics of Monitoring Commands

This section describes semantics of commands used to instrument an original program (say P) with functionality that enables information leak detection.

The operational semantics of monitoring commands (shown via Figure 5.4) is defined as a relation $\rightarrowtail_m$: $(Comm_m \times \mathscr{P}(\mathbb{N}) \times \mathscr{P}(\mathbb{N})) \rightarrow (Comm_m \times \mathscr{P}(\mathbb{N}) \times \mathscr{P}(\mathbb{N}))$ on configurations $\langle c_m : H_{var}, H_{val} \rangle$ and *abort*, where c_m is a monitoring command belonging to the set of monitoring commands $Comm_m$, $H_{var} \in \mathscr{P}(\mathbb{N})$ represents the set of tracked safe addresses and $H_{val} \in \mathscr{P}(\mathbb{N})$ represents the set of tracked secret values. Configuration $\langle \texttt{skip} : H_{var}, H_{val} \rangle$ is final. *abort* denotes a special configuration that leads to an immediate program termination.

In the following, the operational semantics of monitoring commands are discussed.

HvarDef: $$\frac{}{\langle \mathbf{def}(H_{var}) : H_{var}, H_{val}\rangle \rightarrowtail_m \langle \texttt{skip} : \emptyset, H_{val}\rangle}$$

HvalDef: $$\frac{}{\langle \mathbf{def}(H_{val}) : H_{var}, H_{val}\rangle \rightarrowtail_m \langle \texttt{skip} : H_{var}, \emptyset\rangle}$$

HvarAdd: $$\frac{}{\langle \mathbf{insert}(H_{var}, a) : H_{var}, H_{val}\rangle \rightarrowtail_m \langle \texttt{skip} : H_{var} \cup \{a\}, H_{val}\rangle}$$

HvalAdd: $$\frac{}{\langle \mathbf{insert}(H_{val}, k) : H_{var}, H_{val}\rangle \rightarrowtail_m \langle \texttt{skip} : H_{var}, H_{val} \cup \{k\}\rangle}$$

SecAssert$_1$: $$\frac{}{\begin{array}{l}\langle \texttt{assert}(\mathbf{exists}(a, H_{var}) \,||\, \mathbf{foundIn}(e, H_{val}) = 0) : H_{var}, H_{val}\rangle \\ \qquad\qquad \rightarrowtail_m \langle \texttt{skip} : H_{var}, H_{val}\rangle \\ (a \in H_{var})\end{array}}$$

SecAssert$_2$: $$\frac{}{\begin{array}{l}\langle \texttt{assert}(\mathbf{exists}(a, H_{var}) \,||\, \mathbf{foundIn}(k, H_{val}) = 0) : H_{var}, H_{val}\rangle \\ \qquad\qquad \rightarrowtail_m \langle \texttt{skip} : H_{var}, H_{val}\rangle \\ (a \notin H_{var} \wedge \forall t \in H_{val} : \mathbf{isSimilar}(t, k) = 0)\end{array}}$$

SecAssert$_3$: $$\frac{}{\begin{array}{l}\langle \texttt{assert}(\mathbf{exists}(a, H_{var}) \,||\, \mathbf{foundIn}(k, H_{val}) = 0) : H_{var}, H_{val}\rangle \rightarrowtail_m \mathit{abort} \\ (a \notin H_{var} \wedge \exists t \in H_{val} : \mathbf{isSimilar}(t, k) \neq 0)\end{array}}$$

Figure 5.4: Operational Semantics of Monitoring Commands

Initialisation

Command **def**(H_{var}) (see Figure 5.4, Rule *HvarDef*) initialises the set of tracked addresses of safe locations to an empty set indicated by the configuration $\langle \texttt{skip} : \emptyset, H_{val}\rangle$. The set H_{val}, which tracks secret values is similarly initialised via command **def**(H_{val}) (Rule *HvalDef*).

Safe Locations and Secret Values

Rule *HvarAdd* (Figure 5.4) shows semantics for monitoring command **insert**(H_{var}, a), used to record an address of a secret location (a) to the collection H_{var} that tracks addresses of safe locations. The update of the monitoring state is given by ($H_{var} \cup \{a\}, H_{val}$). Secret values are similarly recorded to H_{val} (Rule *HvalAdd*).

Security Assertions

To detect information leakage at runtime, assignments in the original program are instrumented with security assertions. Security assertions abort the execution of a

program if leakage via assignment of a secret value to an unsafe memory location is detected. This prevents information leaks before they occur. Operational semantics of a security assertion is shown in Rules $SecAssert_1$, $SecAssert_2$ and $SecAssert_3$ (Figure 5.4). The following discusses how security assertions capture and prevent information leaks at runtime.

By Definition 10 assignment $v := e$ (where v is a variable and e is an expression) leaks sensitive information if a secret value (or a value that is '*close enough*' to a secret value) is assigned to an unsafe location. Let set H_{var} store addresses of safe locations and set H_{val} represent the set of secret values. Let a and k be values, such that a represents the address of a variable v and k is the result of evaluation of expression e. Then, assignment $v := e$ leaks secret information if a is not captured by H_{var} and there exists a secret value in H_{val} (say t) that is '*close enough*' to k. That is, if **isSimilar**(t,k) (where value k denotes the result of evaluation of e) returns a non-zero value.

Syntactically, a security assertion is given by command

$$\texttt{assert}(\textbf{exists}(a, H_{var}) \,||\, \textbf{foundIn}(k, H_{val}) = 0) \tag{5.1}$$

Let symbol || denotes logical disjunction. Function call **exists**(a,H_{var}) evaluates to a non-zero value if value a is stored in the set H_{var} of addresses of safe locations and to a zero value otherwise. Monitoring expression **foundIn**(k,H_{val}) determines whether the value k is '*close enough*' to one of the secret values tracked by the collection of secret values H_{val}. A call to **foundIn**(k,H_{val}) returns 0 if no values '*close enough*' to k exist in H_{val} and a non-zero value otherwise. The similarity between two values is given by the function **isSimilar**, which returns 1 if input values are '*close enough*' and 0 otherwise. That is, an expression **foundIn**$(k,H_{val}) = 0$ that evaluates to a non-zero value indicates that there exist no secret values in H_{val} that are '*close enough*' to the input value k and to zero otherwise.

The execution of a security assertion (see Figure 5.4, Rules $SecAssert_1$, $SecAssert_2$ and $SecAssert_3$) that evaluates the safety of some assignment $v := e$ (where v is a variable which address is given by value a, and e is an expression that evaluates to value k) is as follows. The security assertion first checks whether the address of a variable v (given by value a) belongs to the set of safe locations tracked in H_{var} (indicated by **exists**(a,H_{var})). **exists**(a,H_{var}) that evaluates to a zero value (i.e., a is an element of H_{var}) indicates that v represents a safe location and thus the assignment is safe (by Definition 10). In this case the behaviour of the security assertion is equivalent to `skip` (Rule $SecAssert_1$), which allows the program to continue execution. If **exists**(a,H_{var}) evaluates to zero (which indicates that a memory location given by v is not a safe location), expression **foundIn**$(k,H_{val}) = 0$ is evaluated. This checks whether the value k given by expression e is '*close enough*' to one of the secret values tracked via H_{var}. **foundIn**(k,H_{val}) that evaluates to a non-zero value indicates that the application **foundIn** returned 0, thus no similarities between the input value and the set of secret

$$\textit{Def:}\ \frac{}{\mathbf{def}(v) \rightsquigarrow \mathbf{def}(v)} \qquad \textit{Skip:}\ \frac{}{\texttt{skip} \rightsquigarrow \texttt{skip}}$$

$$\textit{Annotated:}\ \frac{}{\langle v := e\rangle \quad \rightsquigarrow \quad \begin{array}{l} v := e; \\ \mathbf{insert}(H_{var}, \mathbf{addressof}(v)) \\ \mathbf{insert}(H_{val}, v) \end{array}}$$

$$\textit{Asgn:}\ \frac{}{v := e \quad \rightsquigarrow \quad \begin{array}{l} \mathbf{def}(\texttt{temp}); \\ \texttt{temp} := e; \\ \texttt{assert}(\mathbf{exists}(\mathbf{addressof}(v), H_{var}) \,\|\, \mathbf{foundIn}(\texttt{temp}, H_{val}) = 0); \\ v := \texttt{temp} \end{array}}$$

$$\textit{If:}\ \frac{c_1 \rightsquigarrow c_1',\ c_2 \rightsquigarrow c_2'}{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'}$$

$$\textit{Seq:}\ \frac{c_1 \rightsquigarrow c_1',\ c_2 \rightsquigarrow c_2'}{c_1\,;c_2 \rightsquigarrow c_1';c_2'} \qquad \textit{While:}\ \frac{c \rightsquigarrow c'}{\texttt{while}\ e\ \texttt{do}\ c \rightsquigarrow \texttt{while}\ e\ \texttt{do}\ c'}$$

$$\textit{Function:}\ \frac{c \rightsquigarrow c'}{f \triangleq c \rightsquigarrow f \triangleq c'} \qquad \textit{Program:}\ \frac{\tilde{f} \rightsquigarrow \tilde{f}'}{f;\ e \quad \rightsquigarrow \quad \begin{array}{l} \mathbf{def}(H_{var}); \\ \mathbf{def}(H_{val}); \\ f';e \end{array}}$$

Figure 5.5: Transformation Rules

values were identified and therefore the assignment is safe (see Rule *SecAssert*$_2$). Otherwise, (i.e., $\mathbf{foundIn}(k, H_{val})$ evaluates to a non-zero value) the information leak is detected (because value that is '*close enough*' to one of the secret values is transferred to an unsafe location) and the program aborts (see Rule *SecAssert*$_3$).

5.2.3 Transformation Rules

This section describes the set of compositional transformation rules that instrument an original program with statements that capture secret values at runtime and assertions that verify the safety of the program's assignments with respect to the captured sets of addresses of safe locations and secret values. Figure 5.5 shows the full set of transformation rules applied on an input program P. This yields a modified program P' equipped with statements that prevent information leakage by assignment of secret values to unsafe locations. The transformation steps are now discussed in greater detail.

Initialisation

The first step instruments an original program with collections of values H_{var} and H_{val} (see Figure 5.5, Rule *Program*). At runtime, H_{var} will hold memory addresses of safe locations and H_{val} will hold the secret values for a particular run.

Capturing Safe Locations and Secret Values

The second step in instrumentation inserts commands that record secret values and safe locations to H_{val} and H_{var} respectively. This is done for every annotated assignment in the input program (see Figure 5.5, Rule *Annotated*).

To record safe addresses, commands that retrieve addresses of memory locations (via the application of the **addressof** operator on a variable assigned a secret value) are inserted immediately after the annotated assignments. These addresses are then added to the collection H_{var} that tracks safe memory locations. This action indicated by the monitoring command **insert**(H_{var}, **addressof**(v)) in Rule *Annotated*, where v is a variable assigned a secret value. This is followed by the command that records the secret value (assigned to the safe location) to the collection H_{val}. This is given by the command **insert**(H_{val}, v), which appends the collection of secret values H_{val} with the value bound to variable v (Rule *Annotated*).

Enforcing Safety of Assignments

Finally, the program is instrumented with security assertions that enforce safety of non-annotated assignments (Rule *Asgn*). Each such assignment is considered potentially unsafe, as it may transfer a secret value to an unsafe location, resulting in leakage. For each non-annotated assignment $v := e$ (where v is a variable and e is an expression), first a temporary variable `temp` is introduced. The result of evaluation of e is stored in `temp`; this is to avoid evaluating e twice (since program expressions include function calls execution of e can result in a side effect). A security assertion that verifies the safety of the assignment with respect to H_{var} and H_{val} is then inserted. The assignment is safe if the location being assigned a value (i.e., **addressof**(v)) is recorded in H_{var}, or if the value assigned to an unsafe location (stored in `temp`) is not found in the collection H_{val} of secret values recorded for a program run. A detailed discussion on how security assertions verify the safety of assignments is given in Section 5.2.2.

Note that since the aim is to prevent information leakage, the assignment is checked for leakage before it is allowed to proceed with transferring the result of evaluation of e (stored in the temporary variable `temp`) to v. In this way, the failing assertion prevents leakage by aborting the run of a program before the unsafe assignment is executed.

5.2.4 Execution of Instrumented Programs

At runtime, security assertions added to the original program P check the safety of its assignments. A failure of a security assertion is indicative of a prevented information leak. On execution, H_{var} and H_{val} are initialised to empty collections. As execution proceeds, values and addresses for annotated assignments are added to H_{var} and H_{val} respectively. Non-annotated assignments are evaluated with respect to data stored in H_{var} and H_{val}. If an unsafe location is being assigned a value, P' invokes a security assertion that aborts the execution if a secret value is transferred to an unsafe location. A program run that has no detected failures does not leak any secret information via assignments.

5.3 Application to C Programs

The transformation rules in Figure 5.5 are defined for the abstract language, and need to be mapped to a concrete level to apply to a real programming language. Further, implementations need to be provided of the **isSimilar** and **foundIn** functions, operators **addressof** and collections H_{var} and H_{val}. Note that this means that **isSimilar** and **foundIn** can be tailored to suit specific safety requirements. This section discusses how to adapt the present approach for C programs.

Program Annotation

To record secret values for a program run, assignments that transfer secret values to safe locations are annotated at the source level of the C programming language. These annotations are merely serve to instruct the instrumentation engine on the locations of secret values. The programs are annotated by a developer or an analyst, who manually marks assignments that transfer secret data to safe locations. An alternative way of introducing annotations is by marking functions. For example, the C standard library provides functionality to fetch passwords (e.g., `getpass`), encrypt data (e.g., `crypt`), read data from password databases (e.g., `/etc/shadow` via `getspnam` or `getspent`) or read data from standard input or files (e.g., `gets`, `fgets`, `scanf`). Here it is assumed that the values these functions retrieve are secret, and automatically annotate assignments that transfer values retrieved using these functions to program variables. For example, for an assignment `char *pwd = getpass()`, the instrumentation engine generates code to record a secret value pointed to by `pwd`.

Value Containers

To store safe locations and secret values, red-black trees are used. H_{var} stores addresses of safe locations as the start addresses of memory blocks. That is, the elements

of H_{var} are integers wide enough to store memory addresses (e.g., `intptr_t`). H_{val}, which represents a collection of secret values, stores untyped (i.e., `void*`) pointers to copies of secret values and their sizes. Since C is a weakly typed language, these can be typecasted as needed at comparison time.

Secret Values

Variables identified at the source level as ones that hold or point to secret data are used to record addresses of safe locations and secret values to collections H_{var} and H_{val} respectively. The values and locations are recorded in the context of the syntactic types of variables that hold secret data. For example, for a character pointer `char *s` that points to a secret value, its value is given by the application of `strdup(s)`, which creates a copy of the C string pointed to by `s`, and its safe location is `&*s`, which is the start address of a memory block that holds actual data. Note that in C, `&*s`, which explicitly extracts the address of the memory block pointed to by `s`, evaluates to same address as `s`. Composite types (i.e., structs) are processed with respect to their elements. For example, `struct stt { char *p1, char *p2 } st` has memory locations `&*st.p1` and `&*st.p2`.

In a more complex but typical scenario, one needs to take into account dynamic memory allocation in order to calculate secret values. This is because it is not always possible to determine the size of a memory block statically. For example, given a double-pointer `int **p`, which points to some set of secret values, one needs to know the size of `*p` in order to retrieve all the values to which it points to. This type of information can be determined by tracking memory allocation.

Library or External Functions

Functions for which source code is available are instrumented as described above. However, approximations are required for the library, or external functions, for which the source code is not available. Calls to external functions are potential sources of information leakage because function calls may leak their arguments. For example, `printf("%s",ptr)` is a security violation if `&ptr[0]` is a safe location. Such functions are annotated before the analysis as either *safe* or *unsafe*. Every call to an unsafe external function is instrumented with an assertion that evaluates that function's arguments and fails if any of the arguments leak secret values. For example, a call to the unsafe standard function `printf("%s",ptr)` is transformed to `if (`**addressof**`(ptr)` $\in H_{var}$`) assert(0); printf("%s",ptr);`.

In C external function calls can make assignments to their pointer arguments. For example, library function `strcpy(char *dest, const char *src)` copies the value of `src` to `dest`. For safety reasons, it is over-approximated that any parameter of an external function that is a pointer is assigned a value within the body of that function.

The program is analysed as if there were an assignment of that parameter's value to itself immediately after the execution of the function. For example, for a function call `strcpy(dest,src)` it is assumed that `dest` is assigned in the body of `strcpy`, and treat the call as if it were `strcpy(dest,src); dest = dest;`. These transformations need to be performed before the instrumentation to track the values.

Value Comparison

The present studies primarily target discovering information leakage via disclosed or partially disclosed strings. Thus, rather than using a direct look-up based on equality **foundIn** is implemented as the function that uses the Levenshtein distance [147] as the measure of similarity (encapsulated in the **isSimilar** function) between values as strings. The Levenshtein distance is frequently used to evaluate the strength of passwords against dictionaries [148]; it is computed by counting the number of edits required to transform one string into another. **foundIn** detects that strings are '*close enough*' if the Levenshtein distance between strings is less than a pre-defined threshold, and identifies strings as different otherwise. The benefit of using this measure is that detects similar, but not identical strings. For example, this can detect the leakage of a password *secret string*, via a partially exposed string *'secret trunk'*, which can be converted to *'secret string'* using three edits. Failures due to comparison of strings of less than a threshold length are solved using an identity function as a measure of string similarity.

Safe Termination

C memory de-allocation procedures (e.g., `free`) do not guarantee the destruction of the values stored in de-allocated memory regions. This may result in disclosure of secret values left in memory after a program terminates. The present analysis checks that a program correctly cleans up its secret values by first disabling memory de-allocation functions, and then, before program termination, checking that no location in H_{var} points to a value from H_{val}. This check is triggered using standard C library functions `atexit` and `signal`. Note that while this approach is adequate for the experimentation purposes, a more sophisticated approach could be used in production monitoring systems; for example, one that remembers designated safe memory addresses and scans possibly freed memory on termination.

5.4 Experimental Results

The present approach for the detection of information leakage has been implemented in a prototype tool for C programs. The research prototype is built on top of the Clang [144] compiler infrastructure. To evaluate the applicability of the present

approach the prototype was used to monitor safety of assignments in real security-oriented software and benchmarked code. This section reports the results of the experimentation that focuses on the trade-offs of using the present approach with respect to the runtime overheads incurred by the monitoring for information leakage.

5.4.1 Objectives

This section discusses the two main objectives of this experimentation.

The first objective is to investigate the runtime overheads of the present approach for an application-specific problem in real software. This aims to address a scenario where a security property requires checking of only a limited number of assignments, for example, only assignments that belong to specific functions or modules, where the rest of the assignments in the program are known to not leak.

To evaluate the approach for an application-specific property, the prototype was used to check the safety of password flow in six well-known security UNIX utilities: `su`, `sudo`, `passwd`, `dropbear`, `ftp` and `vlock`. During this experiment, assignments that transfer password values to program variables are annotated manually. Further, the safety of assignments in functions involved in password authentication is checked. Since applications often use relatively few values and assignments to authenticate a user, the prototype is expected to have near negligible runtime overheads. However, as this analysis checks all assignments involved in the handling of password values, it is also expected to soundly identify password leakage. This experiment is described in greater detail in Section 5.4.3.

The second objective is to investigate the runtime overheads of the present approach for a class of security properties in large and computationally intensive programs. This involves application of the present approach to a scenario where security properties require tracking large sets of secret values and safe locations and also require checking the majority of program assignments for leakage.

Section 5.4.4 discusses the investigation of the application of the proposed technique in information leakage detection to multiple security properties and large programs. Since various security requirements exist, the author chooses to check programs against the security properties from the CWE repository. This is because the CWE repository provides realistic security properties applicable to a wide range of programs, where the attacker model is known and well understood. The present experiments involve monitoring programs for a number of CWE properties related to information leakage, including information leaks via the de-allocated but not cleared out memory; improper handling of sensitive data; exposure of sensitive information through standard output channels; and information leaks via temporary files and file handles. This experiment is divided in two parts. In the first part computationally intensive programs selected from the SPEC CPU [40] datasets are monitored. This aims to evaluate overheads of the present approach for large programs with heavy

workload. The second part of this experiment monitors executions of test suites of security-oriented applications, such as `openssh` or `ccrypt`, for leakage. These runs are used to evaluate the overheads incurred by the present approach in a realistic setting.

For the experimentation with CWE security properties, overheads higher than the overheads of the experimentation with the flow of passwords are expected. This is because CWE properties require tracking of a larger number of safe locations and secret values. However, the overheads can still be expected to be within an acceptable range use for detection of information leaks during testing.

The following sections describe the experimentation in greater detail.

5.4.2 Experimental Setup

This experimentation involved performing series of runs of instrumented and original programs and calculated overheads relative to the execution time of the original programs. To account for variance due to external factors, such as the test automation process or system I/O, the overheads are calculated using the mean over 50 runs of the modified and original executables. In the experiment with password flow a single measurement is taken by executing a program 1000 times. This is because the execution time of a single run of a program from the set of UNIX utilities used is too small to measure accurately. During the experimentation with the CWE properties, where program runs are substantially longer, a single measurement accounts for a single run of a program or a run of a test suite (if available).

To correctly calculate overheads incurred by the prototype implementation, *abort* statements in the instrumented assertions are disabled. The assertions are evaluated and the violations are reported, but program runs are not terminated. The purpose of this is to evaluate the total runtime overheads of applications that trigger failures of the security assertions.

The number of added annotations and instrumented assertions are also reported. This is to determine the extent to which the instrumentation influences overheads incurred by the monitored execution.

The platform for all results reported here was dual-processor 2.4GHz Intel Xenon machine with 16GB of RAM, running Gentoo Linux.

Since different experiments use different annotations, security properties, programs and input values, these details are discussed in separate sections for each experiment.

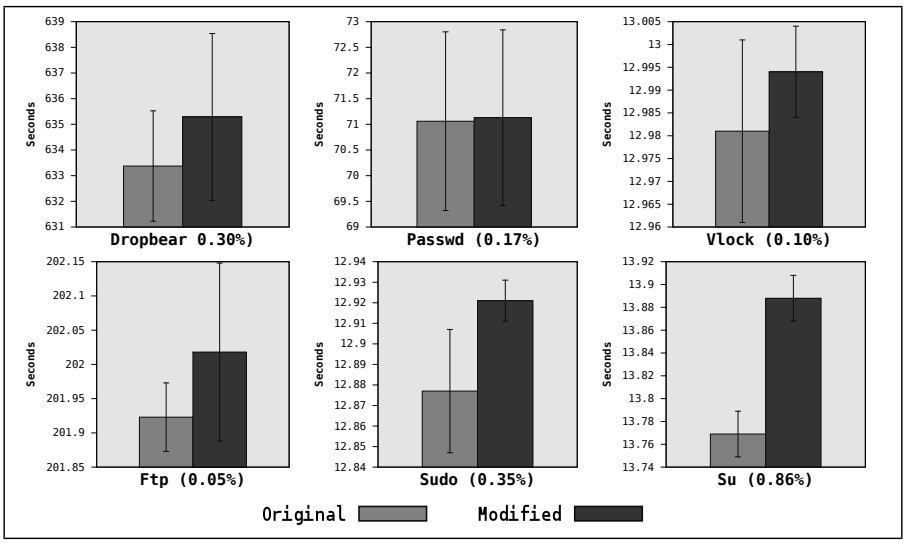

Figure 5.6: Runtime Overheads of UNIX Utilities

5.4.3 Password Flow

This section investigates the application of the author's technique to a problem that requires checking only a limited number of potentially unsafe assignments. The approach is evaluated by checking the flow of passwords in six UNIX utilities: `su`, `sudo`, `passwd`, `ftp`, `vlock`, and `dropbear`. Programs were chosen for this experimentation based on availability of password-authenticating functionality at a system level – tools for which securing password values is important. These tools are mature and have been extensively tested, yet leaks were found in `su` and `ftp`: `su` does not safely destroy the hash sum of the plain-text password, and `ftp` does not overwrite a pointer where the plain-text password value received from the user is stored. For this experiment program locations that transfer password values to program variables were determined manually and the programs used were annotated by hand.

Figure 5.6 shows the running times of the original and instrumented programs. The figure adjacent to the name of the program is the percentage overhead. The bars indicate standard deviation.

In continuous runs of the modified and original versions of the tools correct password values of 10 characters in length were used. The purpose of this is to avoid failures due to short strings comparison, and ensure that the run of a modified program invokes the majority of the instrumented assertions.

It can be seen that the overheads produced by the application of the author's technique do not exceed 1%, and range from 0.05% for `ftp` to 0.86% for `su`. For `dropbear` and `ftp`, however, the standard deviation is considerably greater for the instrumented program than for the original. This is because of variable response times of the network, since during the experimentation both tools were configured to connect to real servers.

Table 5.1 shows the number of source code annotations and injected assertions per

Program	Version	Annotations	Assertions
su	(coreutils 8.13)	3	3
sudo	(1.8.2)	1	14
passwd	(shadow 4.1.4)	3	64
vlock	(2.2)	1	12
ftp	(inetutils 1.8)	2	99
dropbear	(2011.54)	1	5

Table 5.1: Instrumentation Statistics of UNIX Utilities

program. It can be seen that the number of generated assertions does not impact the overhead produced by the instrumentation. For example, the overhead of monitoring execution of `ftp` (0.05%) is the lowest, despite having the largest number of injected assertions (99). This is because the main factors affecting overhead are the length of the strings that need to be compared and the number of secret values tracked.

The results of this experiment suggest that when applied to narrow and well-defined problems such as ensuring the safety of password flow, the proposed technique scales well for real software. The low overhead of the technique is mainly due to instrumentation that checks only assignments directly involved in handling password values. Such an approach results in a lightweight, but sound analysis. This is demonstrated by reporting information leakage discovered in `ftp` and `sudo`, where issues were detected using only a few assertions (e.g., 14 assertions for `sudo`).

It can be noted that low overheads of the prototype for password analysis are attributed to security assertions that evaluate only authentication functionality, where an injected assertion is run once per an input password value. Thus, monitoring in the presence of different properties, where assertions are invoked multiple times (e.g., if placed into the bodies of loops), is likely to result in greater overheads.

The next section reports the results of experimentation with a class of well-known security properties in computationally intensive runs of large software.

5.4.4 CWE-based Security Properties

This section describes the experiment that investigates information leakage using several CWE security properties and large, computationally expensive programs. This experimentation uses the following CWE properties:

- Use of hardcoded or storage of plain-text passwords (CWE-256, CWE-259).
- Exposure of sensitive information via shell messages (CWE-497, CWE-535).
- Software failure to fully clear previously used information in a data structure, file, or other resource (CWE-226).

Program	Dataset	Annotations	Assertions	Program	Dataset	Annotations	Assertions
164.gzip	CPU2000	258	377	*300.twolf*	CPU2000	2375	2986
175.vpr	CPU2000	1270	1872	*401.bzip2*	CPU2006	127	229
176.gcc	CPU2000	3976	5516	*429.mcf*	CPU2006	46	83
177.mesa	CPU2000	1312	1645	*433.milc*	CPU2006	642	1001
179.art	CPU2000	65	104	*456.hmmer*	CPU2006	3620	4077
181.mcf	CPU2000	47	84	*458.sjeng*	CPU2006	1175	1418
183.equake	CPU2000	92	162	*462.libquantum*	CPU2006	62	120
186.crafty	CPU2000	1999	2835	*464.h264ref*	CPU2006	884	1155
188.ammp	CPU2000	1289	1431	*470.lbm*	CPU2006	28	40
197.parser	CPU2000	986	1299	*482.sphinx3*	CPU2006	2814	3057
255.vortex	CPU2000	2185	2278	*998.specrand*	CPU2006	7	14
256.bzip2	CPU2000	123	222	*999.specrand*	CPU2006	7	14

Table 5.2: Instrumentation Statistics of Programs from SPEC CPU Datasets

- Failure to properly clean up and remove temporary or supporting resources after they have been used (CWE-459).

For this experiment, in addition to manual annotations that capture flow of password values into safe locations, the author automatically added annotations that marked memory blocks assigned data received via standard input channels (i.e., using `stdio.h` functions) as safe locations.

The experimentation with the above CWE properties first investigates the overheads of the present approach in computationally expensive runs of large software using programs chosen from the SPEC CPU datasets. Then, the execution of test suites of real security software: `openssh` and `ccrypt` are monitored.

The following section discusses the results of the experiment with CWE properties. It first discusses the experimentation with programs taken from the SPEC benchmarks, and further reports on the results of monitoring of test suites of real software.

Programs from SPEC CPU Datasets

To evaluate runtime characteristics of the present approach on large software C programs selected from the SPEC CPU2000 and CPU2006 were instrumented and monitored. Even though the SPEC benchmarks are not security related, they are used to calculate the overheads on large programs that have a heavy workload. This serves purely to estimate the overheads in extreme situations. Results are reported for all C programs from these sets, except for `253.perlbmk`, `400.perlbench`, `445.gobmk` and `403.gcc`, which were omitted due to compile issues with the original versions of the programs. The sizes of the analysed software ranges from 49 to 140,000 of lines of code (excluding commented and white space lines). Note that these programs have no concern for information security. Thus, information leaks, discovered during the

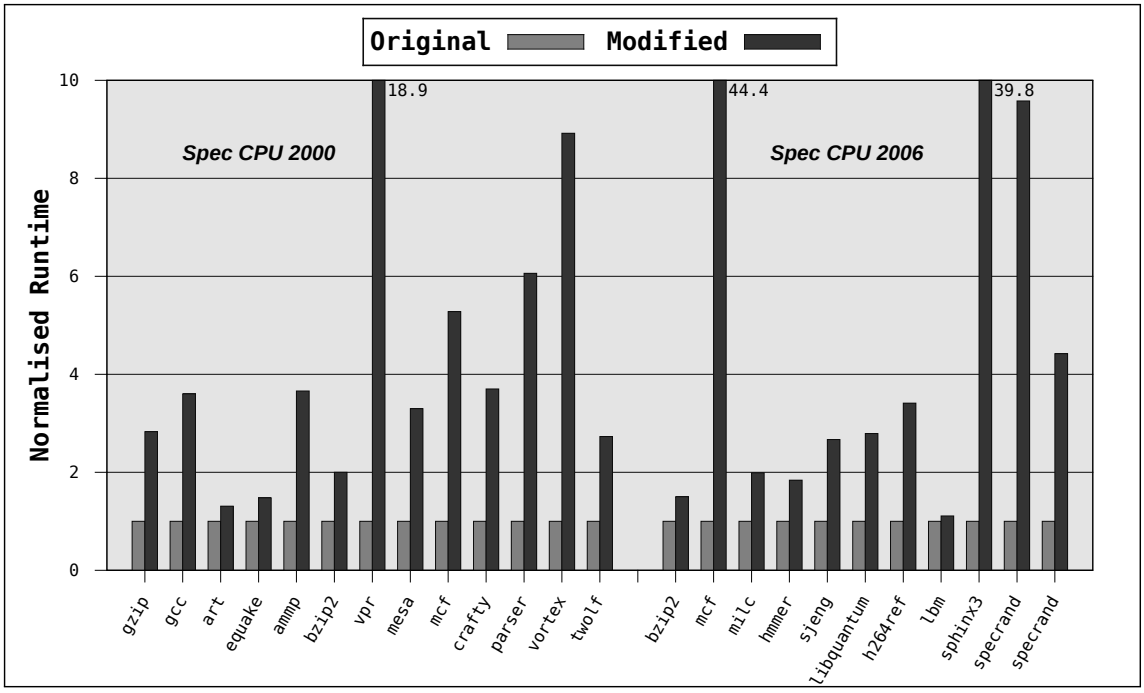

Figure 5.7: Runtime Overheads of Programs from SPEC CPU Datasets

experimentation are unlikely to be of interest to SPEC. However, analysis of such programs is likely to result in many instrumentations and multiple invocations of security assertions. Below are reported the results of the experiments with programs chosen from the SPEC CPU datasets focussing on the runtime overheads caused by the monitored execution.

Figure 5.8 shows the runtime overheads of the prototype relative to the normalised execution time of unobserved runs for the `test` dataset provided by SPEC. For reasons of scale of the figure the exact average times of runs and standard deviation are not presented. This is to better summarise the overhead results for multiple programs, for which runtime varies. The exact runtimes for each program are available from the author on request.

On average, the runtime overheads of the prototype are approximately 7.4 times compared to the normal execution time, with the highest overhead 44.38 times in `429.mcf` and the lowest result of approximately 1.11 times in the `470.lbm` program. The high average time is mainly due to spikes, such as, 39.83 times compared to unobserved execution in `482.sphinx` or 44.38 times in `470.lbm`. In the majority of the programs overheads do not exceed 4 times compared to unobserved execution. The experimentation suggests that the main causes of the runtime overhead are operations on strings in executions of instrumented assertions. That is, the overheads mainly depend on the lengths of the strings checked for leakage, the complexity of the string comparison function (only the Levenshtein distance was used) and the number of assertions executed by the instrumented program.

Further, the experimentation suggests that the runtime overheads of the monitored execution do not depend on the number of annotations or instrumentations

Program	Version	Annotations	Assertions
openssh	(6.2p1)	5421	5839
ccrypt	(1.10)	496	635

Table 5.3: Instrumentation Statistics of Security Programs

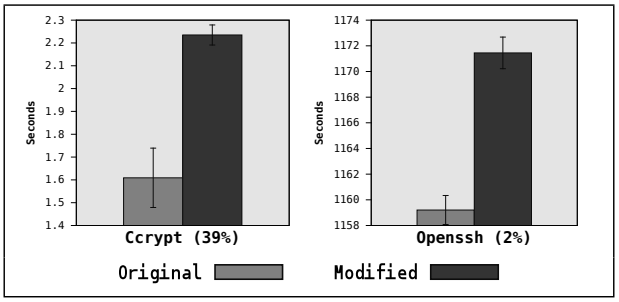

Figure 5.8: Runtime Overheads of Security Programs

(see Table 5.2 for instrumentation statistics for programs selected from the SPEC CPU datasets). This is shown in the differences in runtime overheads between the `176.gcc` and `429.mcf` programs. While `176.gcc` has a high number of assertions (5516), it incurs less than 4 times overhead, whereas `429.mcf`, which uses only 83 assertions, incurs the highest overhead of over 40 times. Such abnormal overhead is due to the security assertions executed multiple times on large value sets, which is directly linked to the number of string comparisons executed by programs. For example, during its execution, `429.mcf` performs over 25 million string comparison operations, whereas `176.gcc` invokes the comparison function less than 200,000 times. Since input to programs for this experimentation is considered secret information, large inputs result in large numbers of values that need to be checked every time a security assertion is invoked. Thus, abnormal cases with high overhead (such as `429.mcf`) should be attributed to the large input to programs requiring checking for leakage.

Experimentation with programs taken from the SPEC benchmarks investigated the overheads incurred by the prototype for the case using large programs and inputs. The next section reports the results of experimentation on security related applications for the same set of CWE-based security properties.

Security Software

This section discusses the results of experimentation with the prototype implementation of the present approach to information leakage detection run on `openssh`, the popular SSH client in UNIX environment, and `ccrypt`, an encryption utility similar to the UNIX `crypt` program. Since overhead introduced by the monitoring techniques depends on input values, security programs that have test suites are chosen. Thus, the number of paths followed for overhead calculation depends on the inputs used by the published test suites.

Figure 5.8 shows the runtime overheads incurred by the monitored execution of test suites of `openssh` and `ccrypt` – 2% and 39% respectively (vertical bars indicate standard deviation). The low overheads of the security tools used are due to relatively low number of input and output operations, compared to the programs selected from the SPEC CPU datasets. For example, the overheads of `openssh` are mainly caused by a number of constants used by the program (i.e., check for CWE-259). Additionally, as data read from configuration files are interpreted as secret, they also contribute to the overhead incurred by the monitored version of the program. Note that, due to a large number of invocations of the monitored executable in the test suite (1090 times for `openssh`), the overhead in standalone invocations will vary slightly and depend, for example, on the size of the configuration files read by the application during a run. Runtime overheads of `ccrypt` are higher (39%); however are still substantially lower than the overheads of the programs taken from the SPEC CPU benchmarks. In `ccrypt` the main cause of overheads is the size of input data interpreted as secret.

Table 5.3 shows instrumentation statistics for `openssh` and `ccrypt`. This is similar to the results of programs selected from the SPEC CPU datasets, where the number of instrumentations was not directly linked to the overhead incurred. It can be seen that monitoring of `openssh` (instrumented with 5839 assertions) results in lower overheads than monitoring of `ccrypt` (instrumented with only 635 assertions).

In summary, the results of the experimentation with CWE vulnerabilities suggest that the overheads of the prototype do not depend on the number of instrumentations; rather, they mainly depend on the size of the input and the number of string comparisons.

5.4.5 Threats to Validity

This section discusses factors that may have affected the validity of the present results.

The first factor is the choice of input data used in the experimentation. For example, insignificant runtime overheads in the experiment with password flow are partially attributed to the number and size of input values. While typical values of passwords were chosen, there is no guarantee that these passwords exercised all paths within programs. No coverage metric have been implemented; thus there could be paths that lead to higher overheads. Small runtime overheads in `openssh` and `ccrypt` are partially attributed to a large number of invocations of monitored executables (1090 and 403 times respectively), where overhead incurred by the program is amortised by operations that are not relevant for information leakage monitoring (e.g., establishing a network connection). Larger overheads may be expected during single executions. Additionally, even though during experimentation with `openssh` and `ccrypt` the author used test suites associated with the utilities, there is no evidence that applying this technique on different programs or using different input values will yield similar results. Finally, during the experiment with programs selected from the

SPEC CPU datasets, the input values provided by SPEC may not be representative for evaluating overheads associated with the detection of information leakage. This is because SPEC concentrates on performance evaluation, rather than on exploring various behaviours.

The second factor involves the choice of programs. The experimentation with password flow used real security tools; however it cannot be claimed these tools are representative of all programs in the security domain. Further, when monitoring for CWE properties, programs from the SPEC benchmarks were taken. These programs focus on performance evaluation, rather than on information security. Finally, the experimentation with CWE properties uses only two real security programs. This is because the aim was to monitor security software with test functionality, which is rare, since releasing such a test suite may be a security issue.

Another factor that may have skewed the results of the empirical evaluation is the annotation process. During the experiment with password flow programs were annotated manually. The author believes all locations and values relevant to password security are tracked; however, some values or locations may have been overlooked. This could result in different overheads. The experimentation with CWE properties relies on automatically derived annotations, where any kind of input data is interpreted as secret. While this is adequate for experimentation, it is not certain that all input information should in practice be tagged *secret*. Similarly, such automatic annotation may have missed some of the values or locations that need to be protected against disclosure. Thus, a better annotation process may improve the accuracy of the results.

The final factor is the string comparison. Although Levenshtein distance is a reliable criterion, it may not be optimal in all cases. Further, as string comparison is the main source of the incurred overheads, a different string comparison function may yield different results. For example, the Hamming distance is likely to result in smaller overheads, than the more expensive Levenshtein distance criteria used in this experimentation.

5.5 Concluding Remarks

This chapter presented a value-driven approach to the detection of leakage of sensitive information at runtime. The proposed approach works on source programs that have annotations marking the secret values that need to protected against disclosure, and the memory locations assumed to store such data safely. This technique first instruments an input program with statements that capture secret values and safe locations, indicated via annotations. Each potentially unsafe assignment is then instrumented with a security assertion that fails if an information leak is detected. The execution of the modified program monitors assignments in the original program for

information leakage. A program execution that has no assertion failures does not leak the nominated secret values.

The proposed approach has been implemented for C programs. The prototype implementation targets the detection leaks of passwords and also supports the detection of well-defined problems inspired by the CWE vulnerabilities, such as, exposure of sensitive information through standard output channels and information leaks via temporary files and file handles.

Experiments with a number of security-oriented UNIX utilities show that the overhead incurred in detecting leakage of passwords is less than 1%. The prototype implementation of the proposed approach detects vulnerabilities in `ftp` and `su` programs. This result suggests that for specific information security problems (e.g., integrity of password values at runtime), the instrumentation introduced by the proposed approach can be used with release versions of software. Further, the experimentation with chosen programs from the SPEC CPU datasets investigates the overheads of the prototype implementation in the case where large programs perform heavy computations and therefore many security assertions are used. Monitoring programs from the SPEC datasets incur higher overheads, as the costs of monitoring increase proportionally to the number of executed assertions that use expensive string comparison operations. However, when the technique is applied to testing, where correctness of programs is established through inexpensive runs (i.e., experimentation with monitoring `openssh` and `ccrypt` test suites), the overheads of the prototype remain low – 39% and 2% for `openssh` and `ccrypt` respectively.

6

Concise Specification Language for Monitoring

Chapters 4 and 5 described monitoring analyses for specific issues, presenting techniques for runtime detection of memory leaks and disclosure of confidential information. This chapter focuses on a generic approach to monitoring programs aimed at error detection. Before this approach is discussed, an argument is advanced for the usefulness of such a generalisation, explaining and how it addresses the limitations of the existing techniques.

6.1 Need for Generalisation

Implementing monitoring from scratch is a daunting task that involves high development costs. In the author's experience, most of the time used for prototype development is spent on addressing technical details and debugging. This motivated a search for a solution, where a prototype is specified concisely at a high level and the concrete implementation is then generated from such specifications; that is, in which the main effort is directed towards developing the elements of the analysis, rather than on handling technical details or re-implementation of standard features.

Early attempts to support the construction of dynamic analyses [73, 75, 76, 78, 79, 86, 149] implemented interfaces for monitoring programs at the source or instruction level. These tools inspired the development of state-of-the-art analysis architectures,

such as Pin [43], Valgrind [10], Frama-C [112] and LLVM [77]. Typically, specification of a monitoring approach using these frameworks requires instrumentation of implementation-specific code at program locations identified by the user. This affords the power to express a wide variety of dynamic analyses; however, the development overhead is also large. It is noted that the prototype implementations of the monitoring techniques discussed in this thesis were created using the *Clang* [144] compiler architecture belonging to the LLVM project [77] and comprise over 15 KLOC of C and C++ code.

Other solutions make use of BISL [96, 105, 106, 109], which formalise intended behaviours of program components via annotations of source programs. The rich features of such languages support monitoring for a wide class of properties, but require manual annotation of each analysed program. This means that this approach does not separate the monitoring specifications from the program under analysis. As a result, BISL specifications cannot be reused. An additional issue is the size of the analysed programs. Consider, for example, the test subjects used in the experiments reported in this thesis (i.e., UNIX utilities and programs from the SPEC CPU datasets). These programs often consist of tens or even hundreds of thousands of lines of code. This factor is likely to contribute to the complexity of specifying monitoring.

This issue of concise and reusable specifications is addressed via runtime verification [114, 120, 122–124, 150], where implementation-level monitors are generated from specifications commonly given by properties in higher-order logic. The generated code is used to check whether a given property holds during the execution of the instrumented program. Such specifications are known to be compact; however, they are not trivial to specify and are hard to optimise due to the gap between the high-level property specifications and implementation-level monitors generated from the properties. In other words, runtime verification techniques are missing the trackable link between abstract descriptions and implementation details.

Finally, monitoring can be enabled through traces of events generated by a running program. Trace monitors originate from AOP [129, 130] as a generalisation of applying advice (i.e., *extra code*) on *pointcuts* (collections of well-defined program points, such as function calls of variable definitions). A trace monitor captures a history of program events at runtime, and observes them by executing extra code if the captured trace matches a partial trace given by the specification. A typical trace monitor specification consists of a *pattern* that describes a partial trace, and an *action* – code executed on a pattern match.

State-of-the-art trace monitors [116, 132, 133, 138, 151] mainly focus on the expressiveness of patterns, and specify actions using the implementation language of a monitored program. Such an approach is adequate for problems that can be captured by patterns, such as checking whether a particular sequence of events is executed. In this case, most of the monitoring code is generated from pattern specifications and

actions are used for simple tasks, such as aborting an execution. However, even expressive pattern languages often require non-trivial actions to be implemented. For example, SQL injection analysis using PQL [138], which claims the expressiveness of a context-free grammar, requires Java implementations of functions that sanitise input and execute SQL. Similarly, a transfer protocol analysis in Arachne [139] uses C networking functions. Such specifications are inherently implementation specific and verbose, contributing to the development overhead and complicating the specification of the analysis. Consider, for example implementing the author's memory leak analysis as a trace monitor. This does not require sophisticated patterns, as all assignments and memory allocations are being observed; the implementation complexity arises from the definitions of actions that compute points-to relationships on-the-fly.

In summary, despite the large body of work in the area, techniques that aim to address specification monitoring at a generic level have limitations. This mainly refers to issues such as lack of separation concerns (in specifications via behavioural interface languages), verbosity and implementation specificity of actions (in trace monitors) and absence of trackable link between monitoring specifications and generated monitors (in runtime verification).

This chapter addresses such limitations and present an approach that aims to reduce the effort associated with developing monitoring code manually. The author proposes the SFM mechanism for specifying monitoring at a higher level of abstraction. An abstract API showing how to implement the monitor is also proposed.

SFM is designed to be concise and has the ability to express various monitoring analyses, focussing on error detection. Further, although SFM specifications are abstract, they are not "too" abstract, allowing its implementation to be efficient. Finally, SFM specifications are kept distinct from the source code of monitoring applications, thus allowing to monitor different programs using a single specification.

SFM is presented as a trace monitor with a key focus on abstract specifications of actions. This enables users to specify actions at a high level and provides a powerful pattern language over sequences of events. Also described is the *monitoring API*: a collection of functions that encapsulate monitoring tasks (e.g., tracking of source locations).

In support this approach a case study in error detection is presented. This case study demonstrates the expressive power of SFM by example and presents SFM specifications that address monitoring for well-known problems, including of stack overflows, information flow vulnerabilities, resource leakage, and SQL injections. This case study aims to demonstrate SFM's capability of addressing runtime defect detection at different levels of abstraction.

This chapter focuses on issues related to concise and expressive specifications for fault detection in memory safety and information flow security. The key ideas

on how to enable instrumentation and implement monitoring components were discussed previously (see Chapters 4 and 5 for details). The contributions made by this chapter are as follows.

- The design of the SFM language, which describes monitoring at a high level of abstraction.
- A case study in the expressiveness of SFM for fault detection, showing shows the power of SFM by example and presenting four complete monitoring specifications that address well known problems in error detection at a high level.

Contributions presented in this chapter have been accepted for a publication in proceedings of the forthcoming ACM/SIGAPP Symposium on Applied Computing (SAC'15) [152].

The reminder of this chapter is organised as follows. Section 6.2 presents the details of the SFM language, discussing how program events, behaviour patterns and actions are represented in SFM. Section 6.3 describes the monitoring API, a collection of functions that encapsulate monitoring tasks for fault detections. Section 6.4 presents monitoring specifications implemented in SFM. Finally, Section 6.5 offers concluding remarks.

6.2 The SFM Language

This section presents the details of the SFM specification language. First, it informally outlines the abstract model and then goes on to discuss SFM syntax.

6.2.1 Informal Model

A run of a program is modelled as a trace (or sequence) of events. An *event* represents a behaviour of a program, such as a function call. Events can also represent points during program execution, such as the start or termination of the program. Events consist of types and attributes, where *type* denotes the kind of the behaviour the event represents (e.g., the function call) and *attributes* describe its details (e.g., the name and arguments of the called function). For the purpose of this thesis, the discussion focuses on function calls, program inputs and outputs and memory operations.

Events are grouped using patterns. A *pattern* is a template that defines partial orders of events using their types and attributes. It is said that a pattern matches a program trace if at runtime the semantics of the pattern satisfies the order of events in the program trace. Patterns can be thought of as regular expressions over the alphabet of program events.

A monitoring specification consists of *trace monitors* that associate *actions* (executable code fragments) with patterns. At runtime, a trace monitor observes an event

by executing an action if the pattern associated with that action matches a program trace. The trace monitors are specified via a syntactic construct **match** *Pat* **using** *Stmt*, where *Pat* is a pattern, *Stmt* represents code executed when *Pat* matches the program trace, and **match** and **using** are keywords.

The following sections discuss how actions, events and patterns are represented in SFM.

6.2.2 Actions

To represent executable code of actions (specified via **using** clause of trace monitors), the subset of the Perl6 language specification [153] is used. Since such elements are standard, only an overview of the supported features is given here. The full EBNF grammar of the SFM language is presented in Appendix A.

The type system of SFM consists of scalars, lists and hashes. Scalars represent integers, floating point numbers and strings, which are augmented with static types `Int`, `Rat`, and `Str` respectively. Lists (denoted by `List`) represent lists of scalars, and hashes (denoted by `Hash`) represent associative arrays. The types of elements in lists and types of keys and values in hashes are inferred from the context. Variable names are prefixed with syntactic type identifiers (`$` for scalars, `@` for lists and `%` for hashes). Access to and modification of elements of lists and hashes is accomplished using standard Perl notation: that is, list elements are retrieved using `[]` subscript and hash elements using `{}`. For example, `$lst[5]` (where variable `@lst` is of `List` type) accesses an element at position `5` and `$hsh{"k"}` retrieves the value bound to key `"k"` in the hash named `%hsh`.

Variables need to be declared before use. A variable is declared using the `my` keyword followed by the type, and variable name, and potentially an initialiser. For example, the statement `my Int $i = 1;` declares an integer variable named `$i`. Variable declarations are either local (to actions) or global.

Expressions are limited to variable names, composite expressions using unary and binary operators, and function calls. Supported operators include `exists` and `delete` unary operators on lists and hashes. `exists` checks whether a given element exists within a list or a hash and `delete` removes an input element from the structure. SFM also implements a number of standard Perl functions such as **`print`** or **`length`**.

Statements consist of variable declarations, *if-then-else* conditional statements, *while* loops and sequential composition of statements. SFM also supports map statements over hashes. For example, given a hash `%hsh`, statement `map { print $_; } %hsh` outputs all keys of this hash.

6.2.3 Events

Events supported by SFM are shown in Table 6.1. The **Type** column shows types of events (as symbolic identifiers); the **Attributes** column lists event attributes and their syntactic types and the column **Description** offers textual description.

SFM events are designed to support fault detection. Memory events (**`read`**, **`write`**, **`malloc`**, **`free`**, **`def`**, **`undef`**) enable detection of memory-related errors such as buffer overruns or illegal dereferences. Note that the author uses a C-like memory model and differentiate between stack and heap allocations. Such a distinction enables the detection of issues such as stack overflows, where only one type of allocation is involved. Events **`in`** and **`out`** describe interactions with the external environment by capturing inputs a program receives and outputs it produces. These events are relevant for detection of information-flow related vulnerabilities, where one is interested in calculating dependencies between private inputs a program receives and public outputs it produces. Structural events (**`begin`**, **`end`**, **`init`**, **`final`**) allow code to be placed at various program points (e.g., to perform initialisation before the program begins). Finally, internal events **`call`**, **`ret`**, and **`flow`** give SFM the ability to observe executions of program modules, thus allowing detection of issues such as API violations, and to track the flow of data within a program (e.g., for taint analysis). It is noted that the set of events supported by the author's approach can be extended to support monitoring for other issues.

6.2.4 Patterns

Patterns are specified via the **match** clause of trace monitors and can be *basic* or *composite*. Basic patterns are similar to events and consist of types and attributes. A basic pattern matches a program trace if it describes the type and attributes of the most recently generated event in the trace. Following Perl conventions, basic patterns are specified as hash initialisers using the *big arrow* notation, which maps names of attributes to concrete values. For example, a basic pattern that matches a heap memory allocation event `malloc` if the size of the allocated block is 10 bytes is written as `{ type => malloc, size => 10 }`. The value mapped to `type` specifies the type of the event the pattern should match, while attribute `size` initialised to 10 constrains the match to those `malloc` events that allocate blocks of 10 bytes. Attribute names that can be used in the construction of basic patterns are shown via **Attributes** column of Table 6.1. Additionally, the keys `type` and `ref` can be used. The `type` key, initialised to one of the keywords given by the **Type** column of Table 6.1, specifies an event type to match (also shown in the example above). The `ref` key specifies the name of the reference to the event. This can be used to retrieve values of event attributes. For example, in the body of an action attached to pattern `{ type => malloc, ref => pat }`, the start address of the allocated block is accessed using

Type	**Attributes**	**Description**
Memory Operations		
read	`Int addr`	Memory access at address `addr`.
write	`Int addr`	Memory modification at address `addr`.
malloc	`Int addr`, `Int size`	Heap memory allocation of `size` bytes starting at address `addr`.
free	`Int addr`, `Int size`	De-allocation of a heap memory block of `size` bytes starting at address `addr`.
def	`Int addr`, `Int size`	Stack memory allocation of `size` bytes starting at address `addr`.
undef	`Int addr`, `Int size`	De-allocation of a stack memory block of `size` bytes starting at address `addr`.
Interactions with Environment		
out	`Int addr`	Data stored in the memory block starting at address `addr` is output to a stream or a file.
in	`Int addr`	Program input is stored to a memory block starting at address `addr`.
Structural Events		
begin	`Str name`	Initial event of function `name`.
end	`Str name`	Final event of function `name`.
init		Initial program event.
final		Final program event.
Internal Events		
call	`Str name`, `Int posn`, `Int addr`	Memory block starting at address `addr` is used as `pos` argument in function call `name`.
ret	`Str name`, `Int addr`	Memory block starting at address `addr` is returned by a function call `name`.
flow	`Int src`, `Int dest`	Value stored in the memory block starting at address `src` flows to a location in the memory block which start address is `dest`.

Table 6.1: Program Events

`$pat->addr` and its size is retrieved via `$pat->size`.

Composite patterns are constructed from smaller patterns using operators, and match multiple events. Operators for pattern composition have been described in the literature [154]; only a subset supported by SFM is reviewed here. Program trace T matches a disjunction of patterns $x|y$, if T matches either x or y. The sequencing operator `>` specifies non-strict matching for chains of events, such that pattern $x>y$ results in a match if events from x occurred at any stage before y. The immediate sequence operator $\sim$ is similar to `>`; however it specifies a strict sequence relationship, such that in pattern $x\sim y$ events from x should immediately be followed by y events. Additionally, patterns are negated using the `!` operator, and grouped with parentheses.

6.3 Monitoring API

SFM events address the types of issues SFM can address. For instance, tracking memory allocations enables analyses for memory safety properties: for example, to detect a stack overflow error, it is sufficient to track stack memory via events `def` and `undef` and report an error if the size of the stack allocation exceeds some limit (e.g., 8MB, the size of stack allocations used by the `gcc` compiler). Such an analysis, however, has some issues. First, a program, can increase or decrease the size of its stack dynamically. That is, a stack limit other than 8MB is likely to result in false alarms or missed errors. Additionally, the language presented in the Section 6.2 cannot track program locations. That is, the analysis is capable of detecting stack overflows; however it cannot report the program locations where such errors originate. This makes it hard to track the detected errors, and consequently diminishes the value of the analysis.

Such issues can be solved by introducing additional events or attributes that capture desired behaviours. For example, events can be extended to support the `location` attribute that holds program locations. However, capturing locations per event requires that every tracked event stores an additional attribute, which is likely to increase overheads. Note that reporting stack overflow errors requires only a single location that corresponds to the "current" program point during a program's execution. Introducing new events (for example for the purpose of tracking locations), increases the number of events that need to be tracked. This is also likely to contribute to the overheads.

From the implementation perspective, a more feasible solution is an API that encapsulates monitoring components. This is because, instead of unconditional generation of events or attributes, the appropriate functionality is triggered by a function call and invoked only where requested by the specification. Further, monitoring API provides a way to reuse common monitoring tasks (e.g., track memory allocation) across multiple specifications. This results in more compact specifications and allows for

Name	Description
`Int isAllocated(Int $addr)`	Return the start address of the memory block address `$addr` belongs to or `0` if `$addr` does not belong to the memory allocation.
`Int blockSize(Int $addr)`	Return the size of a memory block address `$addr` belongs to or `0` if `addr` does not belong to the memory allocation.
`Int stackSize()`	Stack memory allocation size.
`Int heapSize()`	Heap memory allocation size.
`Int stackLimit()`	The maximal size of the process stack.
`Int getTag(Int $addr)`	Return an integer value associated with a memory block identified by address `$addr`.
`Void setTag(Int $addr, Int $val)`	Associate integer `$val` with a memory block given by its start address `$addr`.
`Void abort(Str $fmt,...)`	Output a message to the standard error stream using a printf-like format string and terminate the execution.
`List depends()`	Return a list memory addresses of blocks used in conditions that lead to the current program point.
`Str location()`	Return a symbolic representation of a currently executed program location.

Table 6.2: SFM API

specifying tracking internally, which is likely to be more efficient. For example, in the author's experience, tracking memory allocations at the SFM level (for example, via associative arrays) has higher runtime overheads than internal memory shadowing.

The core functions of the SFM monitoring API are shown in Table 6.2, where the **Name** column shows the function names and arguments they accept, and the column **Description** offers textual description.

The API functions shown in Table 6.2 facilitate detection of a range of runtime faults. This is demonstrated in Section 6.4, which presents complete specifications for runtime detection of problems such as stack overflows, resource leakage, SQL injections and information flow vulnerabilities. It is noted that an analysis may require functionality that is not provided by the API presented here. For example, detecting locations of leakage (as presented in Chapter 4) requires extracting integer values bound to addresses (i.e., dereferencing). In order to handle such extensions interfacing with the target language is allowed, such that a specification in SFM can call a function implemented in the target language. This is similar to the XS interface that allows a Perl script to call functions implemented in C. For the purposes of this thesis, however, the focus is on the expressiveness of the monitoring API (see Table 6.2).

```
1 match { type => def }
2 using {
3     my Int $MaxStackSize  = stackLimit();
4     my Int $StackSize     = stackSize();
5     if ($MaxStackSize < $StackSize) {
6         abort("Stack Overflow at %s",location());
7     }
8 }
```

Listing 6.1: Stack Overflow Detection

6.4 SFM Examples

This section present four complete specifications that demonstrate how SFM addresses monitoring problems in fault detection at different levels of abstraction. The first example, presented in Section 6.4.1, involves runtime detection of stack overflows, a memory safety issue. The specification in Section 6.4.2 shows how a higher-level problem of information flow vulnerability detection is expressed in SFM. Section 6.4.3 discusses monitoring for a CWE-based vulnerability of resource leakage. Finally, Section 6.4.4 presents a SFM specification aimed at detection of SQL injections.

6.4.1 Stack Overflow Detection

Consider a SFM specification (see Listing 6.1) that consists of a monitor executed on events of type **`def`** that allocate memory on the program's stack (see Listing 6.1, Line 1). The maximal size of the process stack is saved to variable `$MaxStackSize` (Line 3) and is retrieved using an API function `stackLimit`. This ensures that `$MaxStackSize` captures the actual stack size of the running program (which reflects possible compile or runtime changes). Variable `$StackSize` (Line 4) captures the actual stack size of the running program that is retrieved using function `stackSize` which returns the size of the program's stack allocation. The check for a stack overflow error is enabled via the conditional statement (Line 5) that compares the stack limit to the actual stack size and aborts the execution (Line 6) if the size of the program's stack allocation (captured via variable `$StackSize`) exceeds the maximal stack size given via variable `$MaxStackSize`. Note that a call to `abort` at Line 6 also reports the location of the detected error via an invocation of function `location`.

The monitoring specification presented in this section provides an example of an application of SFM for detection of a memory-related issue. Overall, in the presence of the monitoring API, detection of memory-related errors is straightforward. For example, illegal accesses or modifications of memory are detected using the `isAllocated` function, which identifies a memory address as belonging (or not) to a program's memory allocation. To detect heap memory leaks (i.e., analysis similar to one implemented by Memcheck), one tracks memory allocations and associated locations via

events **malloc** and **free**, and then uses the **final** event to report the details (e.g., the sizes and locations) of the leaked memory blocks (allocated using **malloc** but never de-allocated by **free**).

6.4.2 Explicit Information Flow

This section presents a SFM specification for the detection of information flow vulnerabilities based on the analysis developed by Denning [155]. Here it is assumed that the reader is familiar with information flow concepts; the discussion is limited to how such an analysis is represented in SFM.

This specification considers only high and low security levels given by integer values 1 and 0 respectively. The security levels are attached to all variables, represented as start addresses of memory blocks, via the built-in tagging mechanism (i.e., functions `getTag` and `setTag` that retrieve and associate integer values with memory blocks).

An information flow security violation via an explicit flow is represented by the flow of data from a memory block tagged 1 to a memory block tagged 0 (i.e., high to low). For an event of type **flow**, which transfers data from a memory block whose start address is given by the attribute `src` to a memory block whose start address is given by the attribute `dest`, the information flow vulnerability is detected if the security level associated with `src` is greater than the security level of `dest`.

Security vulnerabilities via implicit flows are detected using the *global security context*. At a given program point, the global security context is computed as the greatest security level of blocks used in conditions. For example, in the program fragment `if (p && q) { s; }`, at the point given by statement `s`, the global security context is the maximum of security levels associated with memory blocks represented by variables `p` and `q`. Then, given that variable `Pc` holds the global security context, the flow of data from one memory block to another (given by their addresses `src` and `dest`) constitutes an information flow security violation via an implicit flow if the security level captured by `Pc` is greater than the security level of `dest`. That is, a low location is assigned a value that depends (via a condition) on a high value.

The SFM specification shown in Listing 6.2 implements a Denning-style information flow analysis. To simplify the presentation, it is assumed that memory blocks do not change their security levels at runtime and that some of the memory blocks are already tagged with 1, indicating high security.

This SFM specification consists of a single monitor invoked on events of type **flow**. The specification first computes the global security context via a `map` statement (Line 4). This iterates over all addresses used in conditions (retrieved via function `depends`) and stores the highest (i.e., most private) security level in the local variable `$Pc` declared at Line 3. Two conditional statements (Lines 7 and 10) detect information flow security violations via explicit and implicit flows by comparing values

```
1 match { type => flow, ref => e }
2 using {
3     Int $Pc = 0;
4     map depends() {
5         $Pc = max(Pc,getTag($_));
6     }
7     if (getTag($e->dest) < getTag($e->src)) {
8         abort("Flow Violation at %s\n",location());
9     }
10    if (Pc < getTag($e->dest)) {
11        abort("Flow Violation at %s\n",location());
12    }
13 }
```

Listing 6.2: Information Flow Analysis

```
1 my %FH;
2 match { type => ret, name => "fopen", ref => e }
3 using {
4     $FH{$e->addr} = location();
5 }
6
7 match { type => call,  name => "fclose", pos => 1, ref => e }
8 using {
9     delete $FH{$e->addr};
10 }
11
12 match { type => final, ref => e }
13 using {
14     map %FH {
15         printf("Stream opened at %s leaks\n",$FH{$_});
16     }
17 }
```

Listing 6.3: Resource Leakage Detection

of security levels associated with the addresses of memory blocks that participate in the flow (`$e->src` and `$e->dest`) and the global security context stored in `$Pc`. A discovered violation results in the termination of a program run via abort statements (Lines 8 and 11).

6.4.3 Resource Leakage

This section details a SFM specification (Listing 6.3) that implements a dynamic analysis for a CWE property of resource leakage (CWE-404: Improper Resource Shutdown or Release). This specification checks whether file streams allocated by a C program via invocations of `fopen` are properly released via calls to `fclose`.

The global associative array `%FH` (defined at Line 1) tracks streams allocated via `fopen` and maps addresses of allocated streams to the program locations (as strings)

at which they were opened. The monitor defined at Line 2 tracks the return values of calls to `fopen`. This records addresses of streams returned by `fopen` and associates program locations (retrieved using function `location`) in `%FH` via the statement at Line 4. The monitor (defined at Line 7) tracks the first argument of `fclose` calls (i.e., the addresses of released streams). This removes addresses of streams de-allocated by calls to `fclose` from `%FH` (via `delete` statement at Line 9). Finally, the map statement (Line 14) is executed at program termination (via the pattern at Line 12). This iterates over addresses and locations in `%FH` and reports locations stored via `printf` statement (Line 15). Since the `map` statement (Line 14) is executed immediately before the program's termination, a location in `%FH` (i.e., `$FH{$_}`) represents a location of a leaked stream (i.e., a stream never released using `fclose`).

6.4.4 Detection of SQL Injections

The final example demonstrates an SFM specification for detecting SQL injections using an instance of taint analysis.

Consider a C function `sqlexec (struct DB* db, char *buf)` that is used to execute SQL code in a database. Its first argument `DB *db` holds the database connection, whereas string `char *buf` captures the executed SQL code. To prevent a SQL injection, one needs to ensure that a C string supplied as the second argument to `sqlexec` is either sanitised or not affected by user input (e.g., an internal query).

To detect SQL injections, taint analysis is used. This tags buffers that store data received at input as *tainted*. The *tainted* tag associated with the buffer is removed if that buffer is used as an argument of a call to the `sqlcheck` function, which sanitises the inputs. A SQL injection is detected if the second argument of a call to `sqlexec` is tainted; that is, the code to be executed in the database comes from input and has not been sanitised by `sqlcheck`.

An SFM specification for detecting SQL injections is shown in Listing 6.4. Monitoring API functions `getTag` and `setTag` are used to associate and retrieve taint tags (as integers) with memory blocks represented by their start addresses, such that tainted blocks are tagged `1` and untainted are tagged `0`.

The monitoring specification shown in Listing 6.4 consists of three trace monitors. The first monitor (Lines 1-4) is executed for events of type **`in`** that capture data supplied as program input (e.g., via a standard input stream). This monitor taints each memory block containing input data using the statement (Line 3) that associates tag `1` with start addresses of memory blocks (i.e., `$e->addr`) using the `setTag` function.

The second monitor (Lines 6-11) tracks the flow of data and propagates taint tags (via Line 8). This tags a memory block as tainted (i.e., assigns tag `1`) if it is assigned data from another tainted block. That is, the destination of the flow is tainted if the source of the flow is tainted.

Finally, the monitor (Lines 13-21) is executed for function calls. This first untaints

```
1 match { type => in, ref => e }
2 using {
3     setTag($e->addr,1);
4 }
5
6 match { type => flow, ref => e }
7 using {
8     if (getTag($e->src) eq 1) {
9         setTag($e->addr,1);
10    }
11 }
12
13 match { type => call, ref => e }
14 using {
15    if ($e->name == "santise" && pos == 1) {
16        setTag($e->addr,0);
17    }
18    if ($e->name eq "sqlexec" && $e->pos == 2 && getTag($e->addr) == 1) {
19        abort("SQL injection at %s\n",location()); }
20    }
21 }
```

Listing 6.4: SQL Injection Detection

memory blocks sanitised by calls to `sqlcheck` (via Line 15). This is enabled by tracking the first argument of calls to `sqlcheck`. Further, the monitor checks invocations of `sqlexec` and terminates the execution (via the `abort` statement at Line 19) if the second argument of `sqlexec` (which holds SQL code) is tainted (i.e., tagged with `1`). This represents a detected SQL injection.

6.5 Concluding Remarks

This chapter presented a language called SFM for concise and expressive specification of monitoring at a high level of abstraction. The design of SFM aims to reduce effort associated with the manual development of monitoring code.

The design of SFM follows the design of a trace monitor that represents elements that observe programs at runtime using actions and patterns. A key feature of SFM is that it specifies actions at an abstract level. Additionally SFM includes the *Monitoring API* – a collection of functions that encapsulate various monitoring tasks. Such a design allows for creation of concise, reusable and expressive monitoring specifications.

SFM concentrates on issues related to the dynamic detection of program errors. This accomplished via the SFM monitoring primitives (such as events or API functions), which identify the range of problems that SFM can address. The primitives used in designing the SFM language were developed using monitoring analyses described in Chapters 4 and 5.

The expressive power of SFM has been shown by example; four complete monitoring analyses in defect detection have been presented. The issues addressed include stack overflows errors, resource leakage, SQL injections and information flow security vulnerabilities.

This chapter has not discussed issues related to generating implementation-level monitors from the abstract specifications. This is because parts of the SFM specifications presented in this chapter and the resulting C instrumentations were used in implementations of dynamic analyses in memory safety and information flow security reported in Chapters 4 and 5 respectively. The approximations required to adapt abstract SFM descriptions to the concrete level of the C programming language were discussed in the respective chapters.

Overall, SFM summarises the work in monitoring conducted for this thesis and presents a generalisation of the monitoring techniques presented earlier in Chapters 4 and 5.

7 Summary and Future Work

This thesis investigated aspects of the monitoring of C programs, focussing on developing techniques that lead to acceptable overheads. This thesis first presented monitoring techniques to the detection of memory leaks and leakage of confidential information. The proposed techniques were supported by prototype implementations for C programs. The applicability of the suggested techniques was demonstrated using experimentation with real programs and benchmarked code. Further, this thesis explored the idea of specifying monitoring analyses using abstract specifications and presented the language (called SFM) capable of expressing monitoring for a range of problems in bug detection using concise specifications. SFM was supported by a case study that demonstrated how several well-known problems in bug detection (such as stack overflows of SQL injections) are expressed using compact SFM specifications.

This chapter provides a summary of the contributions made by this thesis, offers concluding remarks and discusses directions for future work.

7.1 Summary of Contributions

7.1.1 Detection of Memory Leaks and Leakage Locations

This thesis first presented a technique for detecting memory leaks (Chapter 4). The key issue addressed by this approach is the detection of program locations where the leaked memory was lost, and thus where it potentially can be eliminated. To detect memory leaks programs are instrumented with statements that track memory allocations and operations that potentially update memory structure (e.g., assignments).

The technique associates each tracked memory block with two types of locations – allocation and usage. An allocation location represents a point in a program at which a memory block associated with the location was allocated on the heap. A location of usage represents a program point at which a given memory block was last reachable via program variables. The allocation locations are assigned only once, when blocks are created on the heap. The usage locations are updated based on the execution of the program. Every time a block containing references is updated, its usage location is also updated to reflect the reachability of the block via program variables. The reachability of the block is determined by dynamically computing the dereference of the block's address space.

The presented approach is tunable. Monitoring can be adapted based on the size of the data blocks. This results in a technique where runtime overheads can be reduced, at the cost of reporting less debugging information without losing precision in memory leak detection.

The proposed technique is supported by a prototype implementation for C programs that was used to monitor test suites of several well-known UNIX utilities (e.g., `find`, `grep`, `diff`, `rcs`) and C SPEC CPU benchmarks. Further, the results of the prototype were compared to the results of state of the art memory debugger Valgrind [14]. The results of the experiments show that when only locations of allocations are detected, the prototype implementation significantly outperforms Valgrind. For example, compared to runs of uninstrumented programs, the memory and runtime overheads of the prototype were on an average 1.15 and 1.8 times, while Valgrind incurred approximately 15 times memory and 30.8 times runtime overheads. To detect locations of leakage, which requires computing dereference of the address space of the memory blocks used by a program, the overheads of the prototype increase proportionally to the sizes of the memory blocks. For monitoring UNIX programs the prototype performed better than Valgrind, which could be attributed to relatively small allocated blocks. However, for SPEC benchmarks, which focus on performance evaluation and thus use large inputs and allocate large amounts memory, Valgrind's performance was significantly better. In some cases the large runtime overheads of the prototype for SPEC benchmarks were reduced via overhead tuning, which excluded large memory blocks from tracking. However, the present form of the prototype does not perform any program analysis; this feature relies on a programmer setting the sizes of data blocks manually.

The superior performance of the prototype implementation of the proposed technique is based on the two features. First, memory blocks are tracked using only their start and end addresses. This significantly reduces the memory footprint compared to Valgrind (whose monitor uses 9 bits of memory to track 8 bits in the original program). Second, the proposed technique determines a block's reachability by dynamically computing the dereference of a block's address space. This does not require

capturing and updating pointer structure as the monitor executes; however, at present such an approach is mainly useful where small inputs are used. This is because dereferencing large memory blocks can lead to high overheads, as indicated by the results of the experimentation.

In summary, the results of the experimentation suggest that for memory leak detection where only locations of allocations are reported, the proposed technique may be used as a replacement for binary instrumentation tools, producing similar results with considerably less system resources. However, for tracking leakage locations, the present technique is mainly useful in the domain of functional testing, where program correctness is established using runs with relatively small inputs.

7.1.2 A Value Tracking Approach to Information Flow Security

Chapter 5 of this thesis presented a monitoring technique for runtime discovery of disclosure, or leakage, of private information used by a program.

Unlike similar techniques, such as information flow or taint analysis, that focus on tracking security levels or taint marks attached to program variables, this approach analyses program values and has the ability to determine whether a value disclosed to a third party leads to an information leak with respect to values considered secret at runtime.

The proposed technique analyses programs with annotations that identify secret data. These annotations are used to instrument an input program with statements that capture secret values. Additionally, each potentially unsafe assignment (i.e., one that discloses a value by storing it in a publicly observable location) is instrumented with a security assertion that verifies the safety of the assignments with respect to captured secrets. A run of an instrumented program records secret data, and for every potentially unsafe assignment, executes an assertion that checks if the assignment has leaked a secret value. A failure of the security assertion is indicative of a prevented information leak.

The proposed approach is supported by a prototype implementation for C programs. To evaluate the approach, the prototype was used to check real UNIX security-oriented utilities (such as `openssh`, `sudo`, `su`) and programs selected from the C SPEC CPU datasets for information leakage.

The results of the initial experimentation indicated that runtime overheads to soundly detect application-specific issues; for example leakage of password values in runs of security-oriented UNIX utilities (e.g., `ftp`, `sudo`, `su`, `passwd`) was less than 1%. Even with such small overheads the prototype implementation found real leaks in `ftp` and `su` programs. The next step of the experimentation monitored large programs from the SPEC datasets and real security programs (`openssh` and `ccrypt`) for information leakage using several CWE security properties related to disclosure of confidential information. The results of experiments with the SPEC programs shown

that the prototype implementation can handle large and computationally expensive programs, but the overheads to detect leaks increase significantly averaging to approximately 7.4 times those of the normal execution (640%). However, when using test suites of real security programs, the runtime overheads remain low. This is shown via 39% overhead for the `openssh` test suite and 2% overhead for the `ccrypt` test suite.

The results of the experimentation suggest that the feature of tracking only a limited number of values whose disclosure constitutes information leakage leads to low runtime overheads and detects leaks soundly. This was demonstrated by the results of experimentation with leakage of passwords, where tracking only password values and relevant assignments resulted in extremely low overheads of under 1%, while also leading to discovery of real leaks in `ftp` and `sudo`. This result is explained by the basis of the approach in dynamically tracking values, meaning that there is no need for it to solve aliasing. Monitoring large and computationally expensive programs from SPEC datasets for CWE vulnerabilities resulted in larger overheads, as monitoring SPEC programs requires a large number of values to be tracked. However, such overheads are still within an acceptable range for use with testing. Finally, when the proposed technique is applied to the domain of functional testing (where relatively small inputs are used), its runtime overheads for same CWE properties decrease significantly (as shown via results of `openssh` and `ccrypt`).

Overall, the results suggest that for specific problems (e.g., integrity of password values at runtime), the instrumentation introduced by the proposed approach may be used with release versions of software. For a broader set of properties, requiring tracking of a large number of values, this technique is a good fit for use with testing.

7.1.3 Common Specification Language

The final contribution made by this thesis was presented in Chapter 6 and constitutes a language for abstract and concise specification of monitoring analysis called Specification for Monitoring (SFM).

While detection of different defects typically requires different implementation techniques, there exist a number of monitoring components that can be reused to enable efficient monitoring analysis. The SFM language represents an effort to reduce the development overheads often associated with the implementation of a monitoring analysis from scratch. In SFM, monitoring components are expressed concisely at a high level of abstraction. The actual implementation-level monitoring code is generated from such descriptions. SFM also addresses issues associated with similar approaches that are either “too” abstract or, on the contrary, use implementation-level details.

The main strength of SFM is employment of a separation of concerns principle,

where semantic issues related to monitoring are kept distinct from the implementation-level details. This approach yields compact specifications, as the implementation details are delegated to the implementation of SFM. Also, even though SFM uses abstraction, it is not as abstract as many specification techniques, and thus retains a measure of efficiency for the implementation of the API. The design of SFM follows that of a trace monitor that defines monitoring using *patterns*, which select program events that need to be observed, and *actions*, which encapsulate the functionality that observes executions of events selected by patterns. In this design, the key focus is on the abstract specifications of actions. This represents a departure from existing trace monitors that focus only on the expressiveness of patterns.

In addition to the details of abstract representation, SFM also describes a *monitoring API*: a collection of functions that encapsulate monitoring tasks (e.g., tracking source locations or capturing memory allocations). This feature relieves users of the need to deal with minor implementation details, or to re-implement well-known paradigms.

In its present form, the focus of SFM is on the design of the specification language and its expressiveness. This question of expressiveness is important, as it describes the types of issues SFM can potentially address. At this stage, SFM focuses on tracking monitoring memory allocations, function calls and flow of data within a program (e.g., data assignments or programs inputs and outputs). Implementations of these components for the C programming language were discussed in detail in Chapters 4 and 5; the aim is to make further use of the efficient implementations developed during the work on monitoring for memory leaks and leakage of sensitive information.

To demonstrate the expressiveness of SFM, several examples were presented. These examples show how well-defined problems in error detection, including such issues as stack overflows, information flow vulnerabilities, resource leakage and SQL injections are encoded using the SFM language.

7.2 Future Work

This section describes potential extensions of the work presented in this thesis.

7.2.1 Improving Overhead Results

The results of the experimentation with the technique for the detection of memory leaks and locations of leakage show that, for large inputs this approach produces overheads of up to 1000 times compared to unobserved execution (which cannot be considered acceptable). Therefore, one potential direction for future work is to improve on those results, adapting this technique for monitoring large and computationally expensive programs. A potential solution to this problem is to reduce the number of tracked assignments by using a light-weight, but sound static analysis to

filter out the assignment statements that cannot leak. Further, the results of overhead tuning indicate that in some cases the overheads can be reduced by excluding large blocks assumed to be "data only" (i.e., that do not contain pointers). At the present stage, this is enabled manually, with a programmer setting the size of the blocks that are not to be tracked. Excluding large blocks automatically, for example, by tracking addresses of program pointers and dereferencing only memory blocks that are known to have references, may improve overhead results.

The question of reducing overheads for cases where large values are used is also of interest in extending the technique for detecting information leakage. One possible way is to improve the annotation process and track only a selected set of values, rather than capturing all program inputs and outputs (as done in the experimentation with SPEC programs). Further, overheads may be reduced by using a different string comparison criterion that is relevant for security, but not as expensive as the Levenshtein distance, which was used to determine if leakage occurs.

7.2.2 Using Different Properties

Another potential direction for future work is monitoring for different defect types. The work with memory leaks has shown that the instrumentation used to detect leaks can also be used to detect different defect types (such as illegal dereferences) with overheads similar to memory leak detection. Illegal dereferences is only one defect of potential interest. This approach could also be adapted to address other memory issues, such dangling pointers or accesses of uninitialised memory.

The approach to information leakage detection presented in this thesis is limited to the analysis of values that leak in their entirety via implicit flows. It would be interesting to extend this approach to handling leakage of parts of values (e.g., bit-by-bit) or detecting information disclosure through explicit flows.

7.2.3 Generating Monitoring Analysis

This thesis considered an approach to expressing monitoring analysis using concise and abstract monitoring specifications. Currently, this work concentrates only on the expressiveness of specifications. A logical extension of this work is a complete implementation of the driver to translate SFM to implementation-level code, program instrumentations, and optimisations to reduce the runtime overhead of monitoring.

A

Grammar of the SFM language

```
Constant    = ? Integer, floating point or string literal ? ;

Type        = 'Int' | 'Float' | 'Str' | 'List' | 'Hash' ;

Identifier  = ? An identifier : [A-Za-z][A-Za-z0-9_]+ ? ;

UnaryOp     = '!' | '-'  ;

BinaryOp    = '+' | '-' | '>' | '<' | '>=' | '<=' | '==' | '&&' | '||';

Event       = 'read' | 'write' | 'malloc' | 'free' | 'def' | 'undef'
            | 'out' | 'in'
            | 'begin' | 'end' | 'init' | 'final'
            | 'ret'  | 'call' | 'flow'
            | 'ref' | 'type'
            ;

Attribute   = 'addr' | 'size' | 'name' | 'posn' | 'src'  | 'dest' ;

Specification = { Monitor } ;

Monitor = 'match', Pattern, 'using', Stmt ;

Pattern = '{', Event, '=>', (Constant, Identifier), '}' ;
```

```
Declaration = 'my', Type, Identifier, [ '=' Expression ] ;

Reference = '$', Identifier, '->', Attribute ;

Variable = '$_'
    | Reference
    | ('$', '@', '%') , Identifier
    | '$' , Identifier ,   '{' Expression '}'
    | '$' , Identifier , '[' Expression ']'
    ;

Expression = Variable
    | Constant
    | Expression, BinaryOp, Expression
    | UnaryOp, Expression
    | ('delete', 'exists'), Variable
    | '(' Expression ')'
    | Identifier, '(', [ Expression, { ',', Expression } ] ')'
    ;

Statement  = Declaration
    | 'map', Statement, ('%' , '@'), Identifier
    | 'if', '(', Expression, ')', Statement, [ 'else', Statement ]
    | 'while', '(', Expression, ')', Statement
    | Variable, '=', Expression, ';'
    | ('return' | 'next' | 'last'), ';'
    | '{' Statement '}'
    ;
```

Listing A.1: SFM Grammar (EBNF ISO/IEC 14977)

B

Acronyms

AOP	Aspect Oriented Programming
API	Application Programming Interface
AVL	Adelson-Velsky and Landis
ATOM	Analysis Tools with OM
BNF	Backus-Naur Form
BISL	Behavioural Interface Specification Language
CCI	Configurable C Instrumentation
CFG	Control Flow Graph
CPU	Central Processing Unit
CWE	Common Weakness Enumeration
DBI	Dynamic Binary Instrumentation
EBBA	Event Based Behavioural Abstraction
EBNF	Extended Backus-Naur Form Form
EDL	Event Description Language
EEL	Executable Editing Library
ECC	Error-Correcting Code
GB	Gigabyte
GC	Garbage Collector
GHz	Gigahertz
IFIP	International Federation for Information Processing
ISO	International Organization for Standardization
JPaX	Java PathExplorer

JML	Java Modelling Language
KLOC	Thousand Lines of Code
LLVM	Low-level Virtual Machine
LOC	Lines of Code
LSL	Larch Shared Language
MB	Megabyte
MEDL	Meta Event Definition Language
MOP	Monitoring Oriented Programming
OS	Operating System
PEDL	Primitive Event Definition Language
PQL	Program Query Language
PTQL	Program Trace Query Language
RAM	Random-Access Memory
SFM	Specification For Monitoring
SPEC	Standard Performance Evaluation Corporation
SQL	Structured Query Language
SSH	Secure Shell

Bibliography

[1] V. T. Chakaravarthy. *New results on the computability and complexity of points-to analysis*. In *Proceedings of the SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 115–125 (ACM, 2003).

[2] K. Vorobyov and P. Krishnan. *Comparing model checking and static program analysis: A case study in error detection approaches*. In *Proceedings of the International Workshop on Systems Software Verification*, SSV'10 (USENIX Association, 2010).

[3] E. Clarke, D. Kroening, and F. Lerda. *A tool for checking ANSI-C programs*. In K. Jensen and A. Podelski, eds., *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176 (Springer, 2004).

[4] C. Cifuentes. *Parfait - a scalable bug checker for C code*. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 263–264 (Beijing, China, 2008).

[5] G. R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Xu, M. Park, E. Kleiman, O. Weiss, A. Wehe, and M. Yahya. *The importance of run-time error detection* (2009). `http://rted.public.iastate.edu`.

[6] D. Engler. *Static analysis versus model checking for bug finding*. Concurrency Theory pp. 1–1 (2005).

[7] K. Vorobyov and P. Krishnan. *Combining static analysis and constraint solving for automatic test case generation*. In *Proceedings of the International Academic and Industrial Conference on Testing – Practice and Research Techniques*, pp. 909–914 (IEEE, 2012).

[8] J. Maebe, M. Ronsse, and K. De Bosschere. *Diota: Dynamic instrumentation, optimization and transformation of applications*. In *Proceedings of the Workshop on Binary Translation* (2002).

[9] J. Newsome and D. X. Song. *Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software*. In *Proceedings of the Network and Distributed System Security Symposium* (The Internet Society, 2005).

[10] N. Nethercote and J. Seward. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 42 of *PLDI '07*, pp. 89–100 (ACM, New York, NY, USA, 2007).

[11] J. A. Clause, W. Li, and A. Orso. *Dytan: a generic dynamic taint analysis framework*. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pp. 196–206 (ACM, 2007).

[12] J. A. Clause and A. Orso. *LEAKPOINT: Pinpointing the causes of memory leaks*. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pp. 515–524 (ACM, 2010).

[13] J. Magazinius, A. Russo, and A. Sabelfeld. *On-the-fly inlining of dynamic security monitors*. Computers & Security **31**(7), 827 (2012).

[14] J. Seward and N. Nethercote. *Using valgrind to detect undefined value errors with bit-precision*. In *Proceedings of the USENIX Annual Technical Conference*, pp. 17–30 (USENIX, 2005).

[15] *Projects using Valgrind*. `http://valgrind.org/gallery/users.html`.

[16] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. *AddressSanitizer: A fast address sanity checker*. In *Proceedings of the USENIX Annual Technical Conference*, pp. 309–319 (USENIX Association, 2012).

[17] T. M. Chilimbi and M. Hauswirth. *Low-overhead memory leak detection using adaptive statistical profiling*. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XI, pp. 156–164 (ACM, New York, NY, USA, 2004).

[18] G. Novark, E. D. Berger, and B. G. Zorn. *Efficiently and precisely locating memory leaks and bloat*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 397–407 (ACM, New York, NY, USA, 2009).

[19] W. Lim, S. Park, and H. Han. *Memory leak detection with context awareness*. In *Proceedings of the Research in Applied Computation Symposium*, pp. 276–281 (ACM, 2012).

[20] J. Maebe, M. Ronsse, and K. D. Bosschere. *Precise detection of memory leaks*. In *Proceedings of the International Workshop on Dynamic Analysis*, pp. 25–31 (2004).

[21] R. Hastings and B. Joyce. *Purify: Fast detection of memory leaks and access errors*. In *Proceedings of the Winter USENIX Conference*, pp. 125–136 (1992).

[22] D. Bruening and Q. Zhao. *Practical memory checking with Dr. Memory*. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pp. 213–223 (IEEE Computer Society, Washington, DC, USA, 2011).

[23] Intel Inspector. *Memory and thread debugger*. `https://software.intel.com/en-us/intel-inspector-xe`.

[24] LeakSanutizer. `http://code.google.com/p/address-sanitizer/wiki/LeakSanitizer`.

[25] H. Boehm. *Dynamic memory allocation and garbage collection*. Computers in Physics **9**(3), 297 (1995).

[26] *A garbage collector for C and C++: Current users*. `http://www.hboehm.info/gc`.

[27] T. H. Austin and C. Flanagan. *Efficient purely-dynamic information flow analysis*. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '09, pp. 113–124 (ACM, 2009).

[28] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. *RIFLE: An architectural framework for user-centric information-flow security*. In *Proceedings of the International Symposium on Microarchitecture*, pp. 243–254 (IEEE Computer Society, 2004).

[29] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. *LIFT: A low-overhead practical information flow tracking system for detecting security attacks*. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pp. 135–148 (2006).

[30] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. *TaintEraser: Protecting sensitive data leaks using application-level taint tracking*. SIGOPS Operating Systems Review **45**(1), 142 (2011).

[31] T. H. Austin and C. Flanagan. *Permissive dynamic information flow analysis*. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pp. 3:1–3:12 (ACM, 2010).

[32] D. Hedin and A. Sabelfeld. *Information-flow security for a core of JavaScript*. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pp. 3–18 (IEEE, 2012).

[33] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. *Jsflow: tracking information flow in JavaScript and its APIs*. In *Proceedings of the Symposium on Applied Computing*, pp. 1663–1671 (ACM, 2014).

[34] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. *Improving application security with data flow assertions*. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pp. 291–304 (ACM, 2009).

[35] M. Assaf, J. Signoles, F. Tronel, and E. Totel. *Program transformation for non-interference verification on programs with pointers*. In *Proceedings of the International Conference on Security and Privacy Protection in Information Processing Systems*, vol. 405 of *IFIP Advances in Information and Communication Technology*, pp. 231–244 (Springer, 2013).

[36] S. Blazy and X. Leroy. *Mechanized semantics for the Clight subset of the C language*. Journal of Automated Reasoning **43**(3), 263 (2009).

[37] X. Leroy. *Formal verification of a realistic compiler*. Communications of ACM **52**(7), 107 (2009).

[38] M. D. Bond and K. S. McKinley. *Tolerating memory leaks*. SIGPLAN Notices **43**(10), 109 (2008).

[39] G. Novark, E. D. Berger, and B. G. Zorn. *Plug: Automatically tolerating memory leaks in C and C++ applications*. Tech. Rep. UM-CS-2008-009, University of Massachusetts, Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (2008).

[40] Standard Performance Evaluation Corporation. *SPEC CPU* (2006). `http://www.spec.org/benchmarks.html`.

[41] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2004).

[42] *Mac OS developer tool manual for leaks*. `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/leaks.1.html`.

[43] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. *Pin: Building customized program analysis tools with dynamic instrumentation*. In *Proceedings of the ACM SIGPLAN*

Conference on Programming Language Design and Implementation, pp. 190–200 (ACM, 2005).

[44] Discover. *Oracle solaris studio 12.2 discover and uncover user's guide*. https://docs.oracle.com/cd/E18659_01/html/821-1784/toc.html.

[45] G. Watson. *Dmalloc – debug malloc library*. http://dmalloc.com.

[46] mtrace. *Allocation debugging*. http://www.gnu.org/software/libc/manual/html_node/Allocation-Debugging.html.

[47] H. Ayguen and M. Eddington. *D.U.M.A. - Detect Unintended Memory Access - A Red-Zone memory allocator*. http://duma.sourceforge.net.

[48] B. Meredith. *Omega: An instant leak detector tool for valgrind* (2006). http://www.brainmurders.eclipse.co.uk/omega.html.

[49] Parasoft Insure++. *Runtime analysis and memory error detection for C and C++*. http://www.parasoft.com/insure.

[50] M. Hirzel and A. Diwan. *On the type accuracy of garbage collection*. In *Proceedings of the International Symposium on Memory Management*, pp. 1–11 (ACM, 2000).

[51] C. Jung, S. Lee, E. Raman, and S. Pande. *Automated memory leak detection for production use*. In *Proceedings of the International Conference on Software Engineering*, pp. 825–836 (ACM, 2014).

[52] M. D. Bond and K. S. McKinley. *Bell: Bit-encoding online memory leak detection*. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pp. 61–72 (ACM, New York, NY, USA, 2006).

[53] F. Qin, S. Lu, and Y. Zhou. *Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs*. In *Proceedings of the International Conference on High-Performance Computer Architecture*, pp. 291–302 (IEEE Computer Society, 2005).

[54] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. *Memtracker: Efficient and programmable support for memory access monitoring and debugging*. In *Proceedings of the International Conference on High-Performance Computer Architecture*, pp. 273–284 (IEEE Computer Society, 2007).

[55] A. Sabelfeld and A. Myers. *Language-based information-flow security*. IEEE Journal on Selected Areas in Communications **21**(1), 5 (2003).

[56] L. Zheng and A. C. Myers. *Dynamic security labels and static information flow control*. International Journal of Information Security **6**(2-3), 67 (2007).

[57] A. C. Myers. *JFlow: Practical mostly-static information flow control*. In *Proceedings of the Symposium on Principles of Programming Languages*, pp. 228–241 (1999).

[58] A. C. Myers and B. Liskov. *Protecting privacy using the decentralized label model*. ACM Transactions Software Engineering Methodology **9**(4), 410 (2000).

[59] V. Simonet. *Flow Caml in a nutshell*. In G. Hutton, ed., *Proceedings of the APPSEM-II Workshop*, pp. 152–165 (Nottingham, United Kingdom, 2003).

[60] J. Yu, S. Zhang, P. Liu, and Z. Li. *LeakProber: a framework for profiling sensitive data leakage paths*. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pp. 75–84 (ACM, 2011).

[61] G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. *Automata-based confidentiality monitoring*. In *Proceedings of the Asian Computing Science Conference on Advances in Computer Science*, vol. 4435 of *ASIAN'06*, pp. 75–89 (Springer-Verlag, 2006).

[62] G. Le Guernic. *Automaton-based confidentiality monitoring of concurrent programs*. In *Proceedings of the Computer Security Foundations Symposium*, pp. 218–232 (IEEE, 2007).

[63] D. Chandra and M. Franz. *Fine-grained information flow analysis and enforcement in a Java virtual machine*. In *Proceedings of the Computer Security Applications Conference*, pp. 463–475 (IEEE Computer Society, 2007).

[64] C. Wang, S. Hu, H. Kim, S. R. Nair, M. Breternitz, Z. Ying, and Y. Wu. *StarDBT: An efficient multi-platform dynamic binary translation system*. In *Proceedings of the Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pp. 4–15 (Springer-Verlag, Berlin, Heidelberg, 2007).

[65] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. *TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones*. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 393–407 (USENIX Association, 2010).

[66] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall. *These aren't the droids you're looking for: retrofitting android to protect data from imperious applications*. In *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 639–652 (ACM, 2011).

[67] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. *Preventing information leaks through shadow executions*. In *Proceedings of the Computer Security Applications Conference*, pp. 322–331 (IEEE Computer Society, 2008).

[68] D. Devriese and F. Piessens. *Noninterference through secure multi-execution*. In *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 109–124 (IEEE Computer Society, 2010).

[69] T. H. Austin and C. Flanagan. *Multiple facets for dynamic information flow*. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 165–178 (ACM, 2012).

[70] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda. *Panorama: Capturing system-wide information flow for malware detection and analysis*. In *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 116–127 (ACM, 2007).

[71] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. *Privacy Oracle: a system for finding application leaks with black box differential testing*. In *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 279–288 (ACM, 2008).

[72] A. R. Yumerefendi, B. Mickle, and L. P. Cox. *Tightlip: Keeping applications from spilling the beans*. In *Proceedings of the Symposium on Networked Systems Design and Implementation* (USENIX, 2007).

[73] A. Srivastava and A. Eustace. *ATOM - a system for building customized program analysis tools*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 196–205 (ACM, 1994).

[74] A. Eustace and A. Srivastava. *ATOM: A flexible interface for building high performance program analysis tools*. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pp. 303–314 (USENIX Association, 1995).

[75] J. R. Larus and E. Schnarr. *EEL: Machine-independent executable editing*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291–300 (ACM, 1995).

[76] R. F. Cmelik and D. Keppel. *Shade: A fast instruction-set simulator for execution profiling*. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pp. 128–137 (ACM, 1994).

[77] C. Lattner and V. Adve. *LLVM: A compilation framework for lifelong program analysis & transformation*. In *Proceedings of the International Symposium on*

Code Generation and Optimization, CGO '04 (IEEE Computer Society, Washington, DC, USA, 2004).

[78] B. Bruegge, T. Gottschalk, and B. Luo. *A framework for dynamic program analyzers*. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 65–82 (ACM, 1993).

[79] B. Bruegge. *BEE: A basis for distributed event environments: Reference manual*. Tech. Rep. CMU-CS-90-180, Carnegie Mellon University, Pittsburgh, PA 15213 (1990).

[80] C. L. Jeffery and R. E. Griswold. *A framework for execution monitoring in Icon*. Software: Practice and Experience **24**(11), 1025 (1994).

[81] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. *A lightweight architecture for program execution monitoring*. SIGPLAN Notices **33**(7), 67 (1998).

[82] C. J. Jeffery. *The alamo execution monitor architecture*. Electronic Notes Theoretical Computer Science **30**(4), 198 (2000).

[83] R. E. Griswold, D. R. Hanson, and J. T. Korb. *The icon programming language: An overview*. SIGPLAN Notices **14**(4), 18 (1979).

[84] K. Templer and C. L. Jeffery. *A configurable automatic instrumentation tool for ANSI C*. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pp. 249– (IEEE Computer Society, 1998).

[85] K. S. Templer. *Implementation of a Configurable C Instrumentation Tool*. Master's thesis, The University of Texas, San Auntonio, Texas, USA (1998).

[86] R. A. Olsson, R. H. Crawford, and W. W. Ho. *A dataflow approach to event-based debugging*. Software: Practice and Experience **21**(2), 209 (1991).

[87] P. C. Bates. *Debugging heterogeneous distributed systems using event-based models of behavior*. ACM Transactions Computer Systems **13**(1), 1 (1995).

[88] E. Jahier and M. Ducassé. *Generic program monitoring by trace analysis*. Theory and Practice of Logic Programming **2**(4-5), 611 (2002).

[89] E. Jahier and M. Ducassé. *Generic and efficient program monitoring by trace analysis*. Computing Research Repository **cs.PL/0311016** (2003).

[90] Z. Somogyi, F. Henderson, and T. C. Conway. *The execution algorithm of mercury, an efficient purely declarative logic programming language*. Journal of Logic Programming **29**(1-3), 17 (1996).

[91] C. Flanagan and S. N. Freund. *The RoadRunner dynamic analysis framework for concurrent programs*. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 1–8 (ACM, 2010).

[92] A. Kinneer, M. B. Dwyer, and G. Rothermel. *Sofya: Supporting rapid development of dynamic program analyses for java*. In *Companion to the Proceedings of the International Conference on Software Engineering*, ICSE COMPANION '07, pp. 51–52 (IEEE Computer Society, Washington, DC, USA, 2007).

[93] A. Kinneer, M. B. Dwyer, and G. Rothermel. *Sofya: A flexible framework for development of dynamic program analyses for java software*. Tech. Rep. TR-UNL-CSE-2006-0006, Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, Nebraska, USA (2006).

[94] Apache Commons. *The byte code engineering library* (2001). `http://commons.apache.org/bcel`.

[95] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. J. Parkinson. *Behavioral interface specification languages*. ACM Computing Surveys **44**(3), 16 (2012).

[96] D. C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying ADA Programs* (Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990).

[97] S. Sankar, D. Rosenblum, and R. Neff. *An implementation of anna*. In *Proceedings of the ACM SIGAda International Conference on Ada*, SIGAda '85, pp. 285–296 (Cambridge University Press, New York, NY, USA, 1985).

[98] S. Sankar and M. Mandal. *Concurrent runtime monitoring of formally specified programs*. Computer **26**(3), 32 (1993).

[99] D. C. Luckham, S. Sankar, and S. Takahashi. *Two-dimensional pinpointing: Debugging with formal specifications*. IEEE Software **8**(1), 74 (1991).

[100] B. Meyer. *Object-Oriented Software Construction* (Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988), 1st ed.

[101] J. V. Guttag and J. J. Horning, eds. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science (Springer-Verlag, 1993). With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[102] G. T. Leavens. *An overview of Larch/C++: Behavioral specifications for C++ modules*. Tech. Rep. TR #96-01e, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011-1040, USA (1996).

[103] D. Guaspari, C. Marceau, and W. Polak. *Formal verification of Ada programs*. IEEE Transactions on Software Engineering **16**(9), 1058 (1990).

[104] Y. Cheon and G. T. Leavens. *The Larch/Smalltalk interface specification language*. ACM Transactions on Software Engineering Methodology **3**(3), 221 (1994).

[105] M. Barnett, K. R. M. Leino, and W. Schulte. *The Spec# programming system: An overview*. In *Proceedings of the COnference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, vol. 3362 of *Lecture Notes in Computer Science*, pp. 49–69 (Springer Berlin Heidelberg, 2005).

[106] G. T. Leavens, A. L. Baker, and C. Ruby. *Preliminary design of JML: a behavioral interface specification language for Java*. SIGSOFT Software Engineering Notes **31**(3), 1 (2006).

[107] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. *An overview of JML tools and applications*. Intenational Journal on Software tools for Technology Transfer **7**(3), 212 (2005).

[108] L. Burdy, M. Huisman, and M. Pavlova. *Preliminary design of BML: A behavioral interface specification language for java bytecode*. In *Proceedings of the International Conference Fundamental Approaches to Software Engineering*, vol. 4422 of *Lecture Notes in Computer Science*, pp. 215–229 (Springer, 2007).

[109] M. Delahaye, N. Kosmatov, and J. Signoles. *Common specification language for static and dynamic analysis of C programs*. In *Proceedings of the ACM Symposium on Applied Computing*, pp. 1230–1235 (ACM, 2013).

[110] N. Kosmatov, G. Petiot, and J. Signoles. *An optimized memory monitoring for runtime assertion checking of C programs*. In *Proceedings of the International Conference on Runtime Verification*, vol. 8174 of *Lecture Notes in Computer Science*, pp. 167–182 (Springer, 2013).

[111] P. Baudin, P. Cuoq, J. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language Version 1.7* (2013).

[112] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. *Frama-C - a software analysis perspective*. In *Proceedings of the Software Engineering and Formal Methods International Conference*, vol. 7504 of *Lecture Notes in Computer Science*, pp. 233–247 (Springer, 2012).

[113] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. *Java-MOP: Efficient parametric runtime monitoring framework*. In *Proceedings of the International Conference on Software Engineering*, ICSE '12, pp. 1427–1430 (IEEE Press, Piscataway, NJ, USA, 2012).

[114] F. Chen, M. d'Amorim, and G. Rosu. *A formal monitoring-based framework for software development and analysis*. In *Proceedings of the International Conference on Formal Engineering Methods*, vol. 3308 of *Lecture Notes in Computer Science*, pp. 357–372 (Springer, 2004).

[115] F. Chen and G. Rosu. *Java-MOP: A monitoring oriented programming environment for Java*. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440 of *Lecture Notes in Computer Science*, pp. 546–550 (Springer, 2005).

[116] S. Goldsmith, R. O'Callahan, and A. Aiken. *Relational queries over program traces*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 385–402 (ACM, 2005).

[117] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. *Runtime assurance based on formal specifications*. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 279–287 (CSREA Press, 1999).

[118] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. *Java-MaC: a runtime assurance tool for Java programs*. Electronic Notes in Theoretical Computer Science **55**(2), 218 (2001).

[119] M. Kim. *Information Extraction for Run-time Formal Analysis*. Ph.D. thesis, University of Pennsylvania (2001).

[120] K. Havelund and G. Rosu. *Monitoring Java programs with Java PathExplorer*. Electrical Notes in Theoretical Computer Science **55**(2), 200 (2001).

[121] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. *The maude system*. In *Proceedings of the International Conference Rewriting Techniques and Applications*, vol. 1631 of *Lecture Notes in Computer Science*, pp. 240–243 (Springer, 1999).

[122] D. Drusinsky. *The Temporal Rover and the ATG Rover*. In *Proceedings of the International SPIN Workshop on SPIN Model Checking and Software Verification*, pp. 323–330 (Springer, 2000).

[123] M. d'Amorim and K. Havelund. *Event-based runtime verification of Java programs*. ACM SIGSOFT Software Engineering Notes **30**(4), 1 (2005).

[124] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. *Rule-based runtime verification*. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, vol. 2937 of *Lecture Notes in Computer Science*, pp. 44–57 (Springer, 2004).

[125] H. Barringer, D. E. Rydeheard, and K. Havelund. *Rule systems for run-time monitoring: from Eagle to RuleR*. Journal of Logic and Computation **20**(3), 675 (2010).

[126] E. Bodden. *J-Lo: A Tool for Runtime-checking Temporal Assertions*. Master's thesis, RWTH Aachen University, Aachen, NRW, Germany (2005).

[127] N. Decker, M. Leucker, and D. Thoma. *jUnit^{RV}-adding runtime verification to jUnit*. In *Proceedings of the International Symposium on NASA Formal Methods*, vol. 7871 of *Lecture Notes in Computer Science*, pp. 459–464 (Springer, 2013).

[128] M. Leucker. *Teaching runtime verification*. In *Procedings of the International Conference Runtime Verification*, vol. 7186 of *Lecture Notes in Computer Science*, pp. 34–48 (Springer, 2011).

[129] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. *Aspect-oriented programming*. In *Proceedings of the European Conference on Object-Oriented Programming*, pp. 220–242 (Springer-Verlag, 1997).

[130] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *An overview of AspectJ*. In *Proceedings of the European Conference on Object-Oriented Programming*, vol. 2072 of *Lecture Notes in Computer Science*, pp. 327–353 (Springer, 2001).

[131] R. Douence, D. L. Botlan, J. Noyé, and M. Südholt. *Trace-based aspects*. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, eds., *Aspect-oriented Software Development* (Addison-Wesley, Boston, 2005).

[132] R. J. Walker and K. Viggers. *Implementing protocols via declarative event patterns*. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 159–169 (ACM, 2004).

[133] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *Adding trace matching with free variables to AspectJ*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 345–364 (ACM, 2005).

[134] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc *: An extensible AspectJ compiler*. Transactions on Aspect-Oriented Software Development I **3880**, 293 (2006).

[135] P. Avgustinov, J. Tibble, and O. de Moor. *Making trace monitors feasible*. SIGPLAN Notices **42**(10), 589 (2007).

[136] V. Stolz and E. Bodden. *Temporal assertions using AspectJ*. Electronic Notes in Theoretical Computer Science **144**(4), 109 (2006).

[137] P. Hui and J. Riely. *Temporal aspects as security automata*. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages*, p. 19–28 (ACM, 2006).

[138] M. C. Martin, V. B. Livshits, and M. S. Lam. *Finding application errors and security flaws using PQL: a program query language*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 365–383 (ACM, 2005).

[139] R. Douence, T. Fritz, N. Loriant, J. Menaud, M. Ségura-Devillechaise, and M. Südholt. *An expressive aspect language for system applications with arachne*. In *Transactions on Aspect-Oriented Software Development I*, vol. 3880 of *Lecture Notes in Computer Science*, pp. 174–213 (Springer Berlin Heidelberg, 2006).

[140] S. McConnell. *Code Complete*. DV-Professional (Microsoft Press, 2009).

[141] H. R. Nielson and F. Nielson. *Semantics with applications - a formal introduction*. Wiley Professional Computing (Wiley, 1992).

[142] G. Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series (MIT Press, 1993).

[143] K. Vorobyov, P. Krishnan, and P. Stocks. *A dynamic approach to locating memory leaks*. In *Proceedings of the IFIP International Conference on Testing Software and Systems*, vol. 8254 of *Lecture Notes in Computer Science*, pp. 255–270 (Springer, 2013).

[144] Clang. *A C language family frontend for LLVM*. `http://clang.llvm.org`.

[145] Common Weakness Enumeration. *A community developed dictionary of software weakness types*. `http://cwe.mitre.org`.

[146] K. Vorobyov, P. Krishnan, and P. Stocks. *A low-overhead, value-tracking approach to information flow security*. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, vol. 7504 of *Lecture Notes in Computer Science*, pp. 367–381 (Springer, 2012).

[147] V. I. Levenshtein. *Binary codes capable of correcting deletions, insertions and reversals*. Soviet Physics Doklady **10**(8), 707 (1966).

[148] W. Ma, J. Campbell, D. Tran, and D. Kleeman. *Password entropy and password quality*. Network and System Security, International Conference on **0**, 583 (2010).

[149] A. Srivastava and D. W. Wall. *A practical system for intermodule code optimization at link-time*. Tech. Rep. WRL Research Report 92/6, Western Research Laboratory, 250 University Avenue Palo Alto, California 94301 USA (1992).

[150] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. *Jass - Java with assertions*. Electrical Notes Theoretical Computer Science **55**(2), 103 (2001).

[151] P. Avgustinov, J. Tibble, and O. de Moor. *Making trace monitors feasible*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 589–608 (ACM, 2007).

[152] K. Vorobyov, P. Krishnan, and P. Stocks. *A concise specification language for trace monitoring*. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing* (ACM, 2015).

[153] *Perl 6 specification*. `http://perl6.org/specification`.

[154] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001).

[155] D. E. Denning and P. J. Denning. *Certification of programs for secure information flow*. Communications of the ACM **20**(7), 504 (1977).