

ABSTRACT

Title of Dissertation: A COMPILER LEVEL INTERMEDIATE
REPRESENTATION BASED BINARY
ANALYSIS SYSTEM AND ITS APPLICATIONS

Kapil Anand, Doctor of Philosophy, 2013

Dissertation directed by: Professor Rajeev Barua
Department of Electrical and Computer Engineering

Analyzing and optimizing programs from their executables has received a lot of attention recently in the research community. There has been a tremendous amount of activity in executable-level research targeting varied applications such as security vulnerability analysis, untrusted code analysis, malware analysis, program testing, and binary optimizations.

The vision of this dissertation is to advance the field of static analysis of executables and bridge the gap between source-level analysis and executable analysis. The main thesis of this work is scalable static binary rewriting and analysis using compiler-level intermediate representation without relying on the presence of metadata information such as debug or symbolic information.

In spite of a significant overlap in the overall goals of several source-code methods and executables-level techniques, several sophisticated transformations that are well-understood and implemented in source-level infrastructures have yet to become available in executable frameworks. It is a well known fact that a standalone executable without any meta data is less amenable to analysis than the source code. Nonetheless, we believe that one of the prime reasons behind the limitations of existing executable frameworks is that current executable frameworks define their own intermediate representations (IR) which are significantly more constrained than an

IR used in a compiler. Intermediate representations used in existing binary frameworks lack high level features like abstract stack, variables, and symbols and are even machine dependent in some cases. This severely limits the application of well-understood compiler transformations to executables and necessitates new research to make them applicable.

In the first part of this dissertation, we present techniques to convert the binaries to the same high-level intermediate representation that compilers use. We propose methods to segment the flat address space in an executable containing undifferentiated blocks of memory. We demonstrate the inadequacy of existing variable identification methods for their promotion to symbols and present our methods for symbol promotion. We also present methods to convert the physically addressed stack in an executable to an abstract stack. The proposed methods are practical since they do not employ symbolic, relocation, or debug information which are usually absent in deployed executables. We have integrated our techniques with a prototype x86 binary framework called *SecondWrite* that uses LLVM as the IR. The robustness of the framework is demonstrated by handling executables totaling more than a million lines of source-code, including several real world programs.

In the next part of this work, we demonstrate that several well-known source-level analysis frameworks such as symbolic analysis have limited effectiveness in the executable domain since executables typically lack higher-level semantics such as program variables. The IR should have a precise memory abstraction for an analysis to effectively reason about memory operations. Our first work of recovering a compiler-level representation addresses this limitation by recovering several higher-level semantics information from executables. In the next part of this work, we propose methods to handle the scenarios when such semantics cannot be recovered.

First, we propose a hybrid static-dynamic mechanism for recovering a precise and correct memory model in executables in presence of executable-specific artifacts such as indirect control transfers. Next, the enhanced memory model is employed to define a novel symbolic analysis framework for executables that can perform the same types of program analysis as source-level tools. Frameworks hitherto fail to

simultaneously maintain the properties of correct representation and precise memory model and ignore memory-allocated variables while defining symbolic analysis mechanisms. We exemplify that our framework is robust, efficient and it significantly improves the performance of various traditional analyses like global value numbering, alias analysis and dependence analysis for executables.

Finally, the underlying representation and analysis framework is employed for two separate applications. First, the framework is extended to define a novel static analysis framework, *DemandFlow*, for identifying information flow security violations in program executables. Unlike existing static vulnerability detection methods for executables, DemandFlow analyzes memory locations in addition to symbols, thus improving the precision of the analysis. DemandFlow proposes a novel demand-driven mechanism to identify and precisely analyze only those program locations and memory accesses which are relevant to a vulnerability, thus enhancing scalability. DemandFlow uncovers six previously undiscovered format string and directory traversal vulnerabilities in popular ftp and internet relay chat clients.

Next, the framework is extended to implement a platform-specific optimization for embedded processors. Several embedded systems provide the facility of locking one or more lines in the cache. We devise the first method in literature that employs instruction cache locking as a mechanism for improving the average-case run-time of general embedded applications. We demonstrate that the optimal solution for instruction cache locking can be obtained in polynomial time. Since our scheme is implemented inside a binary framework, it successfully addresses the portability concern by enabling the implementation of cache locking at the time of deployment when all the details of the memory hierarchy are available.

A COMPILER LEVEL INTERMEDIATE REPRESENTATION
BASED BINARY ANALYSIS SYSTEM AND ITS APPLICATIONS

by

Kapil Anand

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Professor Shuvra Bhattacharya
Professor Manoj Franklin
Professor Angelos Keromytis
Professor Michael Hicks

© Copyright by
Kapil Anand
2013

To the memory of my mother

Acknowledgments

I would like to thank my advisor, Prof Rajeev Barua, for his continuous guidance throughout my graduate studies. He has not only been instrumental in refining my ideas, but has also provided invaluable fresh perspective towards new research problems. I have always enjoyed my long brainstorming discussion sessions with him. A lot of ideas in this dissertation are direct outcomes of our discussions. He has been patient and encouraging in tough situations when we failed to achieve desired results. As a result of his feedback, I appreciate the significance of patience, focus and thorough experimentation in achieving research excellence. His endless rounds of comments on my research work have been pivotal in improving my research and writing abilities.

I would like to thank Prof Angelos Keromytis for his invaluable guidance and discussion towards the later part of my dissertation. His feedback has been instrumental in expanding the envelope of my work in software security. I extend my gratitude to Prof Michael Hicks, Prof Shuvra Bhattacharya and Prof Manoj Franklin for agreeing to serve on my committee and providing feedback in improving the quality of this dissertation.

Being part of a collaborative research effort, I had the privilege of working closely with several of my labmates over the years - Aparna, Khaled, Matt, Jim, Greeshma, Padraig, Mincy, Kungjin, Don, Tim and Fady. I have enjoyed developing systems together as well as discussing ideas with them during our group meetings. I would especially like to thank Khaled for his extensive help in chasing the deadlines

while working together in the lab at odd hours. I would like to thank Aparna for her numerous feedback and revisions to our research papers.

I have been fortunate to have an extensive set of friends during my stay at College Park- Shalabh, Nitesh, Kaustubh, Rakesh, Raghu, Jishnu, Aparna, Harita, Satish, Ayan, Vaibhav, Ishwar, Ravi, Ashish, Srikanth, Karthik, Rashi, Shiti, Himanshu, Osman, Daniel and many more. To say the least, thanks for all the fun! I look forward towards more such amusing times in future. I am also grateful to my close friends from undergrad era - Shivam, Sagar, Prateek, Shalabh, Tushar, Aditya for staying in touch despite the distance.

I would also like to thank the people behind Eppley Recreation Center for providing and maintaining great facilities. A gentle run at ERC was extremely helpful in unwinding several fruitless days (or weeks) of research. I also thank Vaibhav and Khurram for starting Cricket@ece group and keeping us in touch with our favorite sport.

Finally, I cannot adequately express my gratitude to my family for their continuous support and encouragement during this long journey. You define a meaning to everything, thanks for being there.

Table of Contents

List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Motivation for executable analysis	1
1.2 Advantages of executable analysis	3
1.3 Thesis Statement	5
1.3.1 Assumptions behind this work	7
1.4 Contribution of this dissertation	10
1.4.1 Representation	10
1.4.2 Analysis	12
1.4.3 Applications	14
1.5 SecondWrite	16
1.6 Organization of the Dissertation	18
2 Related Work	21
2.1 Binary rewriting	21
2.2 Binary Analysis/Intermediate Representation recovery	24
2.3 Industrial Tools	26
3 Decompilation to compiler level intermediate representation	28
3.1 Introduction	28
3.1.1 Benefits of abstract stack and symbols	33
3.1.1.1 Symbolic Execution	34
3.2 Overview of the framework	36
3.2.1 Disassembler Module	37
3.3 Deconstruction of physical stack frames	39
3.3.1 Representing the local stack frame	39
3.3.2 Representing procedure arguments	41
3.4 Translating memory locations to symbols	46
3.4.1 Motivation for partitions	47

3.4.2	Reaching definition framework	48
3.4.3	Symbol promotion algorithm	49
3.5	Results	52
3.5.1	Static characteristics	55
3.5.2	Un-optimized input binaries	57
3.5.3	Optimized input binaries	58
3.5.4	Impact of symbol promotion	59
3.5.5	Symbolic Execution	60
3.5.6	Automatic Parallelization	62
4	Symbolic Analysis for executables	64
4.1	Introduction	64
4.2	Related Work	67
4.3	Contribution	71
4.3.1	Redundancy elimination	71
4.3.2	Program Parallelization	74
4.3.3	Alias analysis	78
4.4	Overview	80
4.4.1	Memory abstraction	82
4.5	Symbolic Abstraction	83
4.6	Symbolic Value Analysis	87
4.6.1	Intraprocedural Analysis	87
4.6.2	Interprocedural propagation	94
4.7	Dependence Analysis	97
4.8	Value Numbering	100
4.9	Results	104
4.9.1	Static characteristics	104
4.9.2	Value numbering	108
4.9.3	Program parallelization	110
4.9.4	Alias analysis	112
5	Improving memory abstraction	115
5.1	Precise Memory Model	115
5.2	Motivation	116
5.3	Recovering precise memory abstraction	119
5.3.1	Static Computation	120
5.3.2	Dynamic Mechanism	124
5.4	Results	125
6	Information flow security of executables	128
6.1	Introduction	128
6.2	Related Work	133
6.2.1	Static Information Flow Techniques	133
6.2.2	Dynamic Information Flow Techniques	135
6.2.3	Demand-driven Analysis	137

6.3	Overview of the system	137
6.4	Background	139
6.4.1	Memory Abstraction	139
6.4.2	Information Flow Policy	140
6.5	Demand Driven Set	142
6.6	Demand Driven Information Flow Analysis	145
6.6.1	Information Abstraction	145
6.6.2	Analysis	147
6.6.3	Policy Enforcement	150
6.7	Discussion	153
6.7.1	Indirect calls and branches	153
6.7.2	Limitations	155
6.8	Results	156
6.8.1	Vulnerabilities	158
6.8.2	False Positives	161
6.8.3	Scalability	163
6.8.4	Information Flow Leakage	165
6.8.4.1	KeePassX	165
6.8.4.2	thttpd	166
6.8.5	Spec Benchmarks and Coreutils	168
7	Cache Locking	171
7.1	Introduction	171
7.2	Cache Locking Interface	174
7.3	Related Work	175
7.4	Motivation	179
7.5	Theoretical Analysis of Cache Locking	181
7.6	Static Cache Locking	185
7.6.1	Cache Locking Algorithm	187
7.7	Dynamic Cache Locking	194
7.7.1	Program Points	195
7.7.2	Dynamic Locking Algorithm	198
7.8	Implementation	202
7.9	Results	203
7.9.1	Static Cache Locking	205
7.9.2	Dynamic Cache Locking	209
8	Conclusions and Future Work	213
8.1	Future Directions	214
	Bibliography	217

List of Tables

3.1	<i>Benchmarks Table</i>	53
3.2	<i>Corner cases of our analysis.</i>	56
3.3	<i>Improvement in constraints processing with symbol promotion.</i>	62
4.1	<i>Transfer functions for each instruction in a procedure $Func$. Here, s denotes the size of dereference in a memory access instruction.</i>	91
4.2	<i>Applications Table</i>	105
4.3	<i>Parallelization benchmarks</i>	111
7.1	<i>Application Table</i>	204

List of Figures

1.1	<i>Contributions of the dissertation</i>	11
1.2	<i>SecondWrite system highlighted with the contributions of this dissertation</i>	16
2.1	<i>Comparing SecondWrite with other executable tools</i>	22
3.1	<i>Source-code example. Variable names and types in the source-code recovered by LLVM C-backend have been modified for readability.</i>	29
3.2	<i>A small source-code example and its pseudo-assembly code, showing the limitation of existing methods for detecting arguments.</i>	30
3.3	<i>An example showing that variable identification and symbol promotion are different.</i>	31
3.4	<i>An example showing the simplification in symbolic execution constraints with symbol promotion.</i>	35
3.5	<i>Constraints for Fig 3.4(b).</i>	35
3.6	<i>SecondWrite system.</i>	37
3.7	<i>A small pseudo-assembly code. The second operand in the instruction is the destination.</i>	43
3.8	<i>IR of the pseudo-assembly code. SIZE_BAR is size of ORIG_FRAME_BAR, register names are pure IR symbols.</i>	45

3.9	<i>Symbol promotion. Second operand in the instruction is the destination of the instruction.</i>	47
3.10	<i>The reaching definition description. Definitions are propagated across the control flow of program.</i>	49
3.11	<i>Variation of analysis time with lines of code. Outlier program dealII has been omitted for the ease of presentation.</i>	54
3.12	<i>Percentage of original symbolic accesses recovered in IR.</i>	56
3.13	<i>Partition algorithm visualization</i>	57
3.14	<i>Normalized runtime of rewritten binary as compared to its corresponding input version (=1.0) compiled by gcc.</i>	58
3.15	<i>Normalized runtime of rewritten binary as compared to its corresponding input version (=1.0) compiled by Visual Studio.</i>	59
3.16	<i>Normalized runtime of rewritten binary as compared to optimized version (=1.0) compiled by gcc.</i>	60
3.17	<i>Impact of symbol promotion on runtime of rewritten binary v/s unoptimized input binary (=1.0).</i>	61
3.18	<i>Impact of symbol promotion on runtime of rewritten binary v/s optimized input binary (=1.0).</i>	62
3.19	<i>Automatic parallelization</i>	63
3.20	<i>Number of induction variables recognized</i>	63
4.1	<i>(a) A sample C code (b) Corresponding assembly code, the second operand in the instruction is the destination</i>	72
4.2	<i>(a) Value numbering obtained without propagation through memory locations (b) Value numbering with propagation through memory locations</i>	73
4.3	<i>(a) A sample C code (b) Corresponding assembly code, the second operand in the instruction is the destination</i>	75
4.4	<i>(a) Symbolic expressions obtained with no memory propagations (b) Symbolic expressions with memory propagation</i>	77
4.5	<i>A sample assembly code, second operand in the instruction is the destination</i>	79

4.6	<i>Organization of the system</i>	80
4.7	<i>Grammar for symbolic expressions. + and * are standard arithmetic operators, Int is the set of all integers, IR Variables are symbols in the obtained intermediate representation</i>	83
4.8	<i>An example CFG showing the limitations of symbolic expressions for value numbering</i>	102
4.9	<i>Introduction of Phi for removing the limitations of symbolic expressions for value numbering</i>	103
4.10	<i>Symbolic Value Set Visualization</i>	106
4.11	<i>Percentage of symbolic expressions that containing at least one symbolic alphabet propagated through a memory location, out of symbolic expressions for all IR variables</i>	107
4.12	<i>Variation of TOP (\top) data objects with varying size of symbolic value set</i>	108
4.13	<i>Normalized improvement in detection of equivalent computations (No Symbolic analysis = 1.0)</i>	109
4.14	<i>Normalized improvement in removal of redundant instruction (No symbolic analysis=1.0)</i>	110
4.15	<i>Alias analysis results</i>	113
5.1	<i>An example demonstrating the imprecision in the presence of indirect calls, second operand in the instruction is the destination</i>	119
5.2	<i>Data flow rules used to determine stack modifications in a procedure P</i> 121	
5.3	<i>Percentage of procedures with unknown CTIs. The static represents cases when constraint solvers succeed</i>	125
5.4	<i>Additional alocs added as a result of constraint solvers, normalized to original number of alocs</i>	126
5.5	<i>Variables requiring a new symbolic alphabet in presence of additional a-locs</i>	127
6.1	<i>Organization of the system.</i>	138

6.2	<i>Deduction rules for computing Demand Set. Rules constitute a backward analysis, where a conclusion before an instruction is derived based on the premise after the instruction.</i>	144
6.3	<i>Symbolic Information Grammar: Grammar for information flow abstraction. \cup is the union operator, IR Symbols are symbols in the obtained intermediate representation corresponding to the registers in the input executable, intermediate computations and calls to external library procedures.</i>	146
6.4	<i>Rules for Symbolic Information Analysis.</i>	149
6.5	<i>Vulnerabilities discovered in real-world programs.</i>	157
6.6	<i>Code snipped from csplit showing the format string vulnerability. Second operand is the destination in executable code.</i>	159
6.7	<i>Format string vulnerability detection.</i>	161
6.8	<i>Directory traversal attacks.</i>	162
6.9	<i>Size of Demand Set (SR and SM) normalized (=1.0) to all variables and a-locs respetively.</i>	163
6.10	<i>Scalability of demand driven and exhaustive analysis with increasing lines of code.</i>	164
6.11	<i>Spec Benchmarks</i>	167
6.12	<i>Vulnerability detection in Coreutils</i>	168
6.13	<i>Time for analyzing Coreutils</i>	169
7.1	<i>(a) Weighted CFG of a small part of a program. A, B, C and D are instructions of 4 byte each (b) A hypothetical memory layout of the above instructions (c) A dummy 16-byte direct mapped instruction cache. The alphabets at right hand side of each cache line show the instructions which are mapped to the line according to the cache mapping function (d) The execution trace of this part of the program</i>	179
7.2	<i>(a) Number of misses observed for each node with and without locking (b) Locking of node C in set 0 of cache</i>	181
7.3	<i>Static Cache Locking Algorithm.</i>	192

7.4	<i>Example showing (a) a program outline; and (b) its DPRG showing nodes, edges & timestamps (c) modified DPRG nodes and timestamps assuming that execution frequency of proc_C is greater than LIMIT</i>	195
7.5	<i>Dynamic Cache Locking Algorithm.</i>	200
7.6	<i>The Experimental WorkFlow</i>	202
7.7	<i>Percentage improvement in instruction-cache miss rate over cache with no locking for varying sizes of a 2-way set-associative cache</i>	206
7.8	<i>Percentage improvement in instruction-cache miss rate over cache with no locking for different associativities of the cache. The cache size is kept fixed at 4 Kb.</i>	206
7.9	<i>Improvement in execution time of the applications over cache with no locking for varying size of a 2-way set associative cache</i>	208
7.10	<i>Variation of execution time improvement for processors with different clock speeds for a 4kB 2 way set associative cache</i>	208
7.11	<i>Percentage improvement in instruction-cache miss rate, compared with static cache locking, for a 2-way set-associative cache of size 2 kb,4 kb, 8 kb and 16 kb.</i>	210
7.12	<i>Percentage improvement in instruction-cache miss rate with dynamic cache locking over cache with no locking for different associativities of the cache. The cache size if kept fixed at 8 kb.</i>	211
7.13	<i>Improvement in execution time of the applications with dynamic cache locking, as compared with static and optimal static cache locking, for varying size of a 8 kb 2-way set associative cache</i>	211
8.1	<i>Future Directions</i>	215

Chapter 1: **Introduction**

Analyzing and optimizing programs from their executables has received a lot of attention recently in the research community. In recent years, there has been a tremendous amount of activity in executable-level research targeting varied applications such as security vulnerability analysis [42, 137], untrusted code analysis [13], malware analysis [163], program testing [47], and binary optimizations [130, 103].

1.1 Motivation for executable analysis

Traditional source-code analysis frameworks have limited applicability in several scenarios. It has been demonstrated that executable-level tools can eliminate such limitations faced by source-code analysis frameworks. Below, we discuss some of these scenarios.

Absence of source-code: There are several circumstances where the original source-code is not accessible. Some of the most prevalent reasons are listed below:

- IP-protected software
- Third-party library and software components

→ Malicious executables

→ Legacy executables

All such situations require executable-level tools for distinct applications. For example, due to a rapid rise in cyber attacks, there is an increasing need to certify the behavior and uncover vulnerabilities in IP-protected software and commercial off-the shelf software components. The availability of such components only in an executable form has created a huge demand for effective executable analysis tools to achieve this goal [11, 20, 109].

Further, hundreds of malware are being uncovered almost daily which are only available in executable form [109]. Security researchers requires novel tools to understand the behavior of such malware and to develop effective counter strategies.

Various organizations [11] have critical *legacy* applications that have been developed for older systems and need to be ported to future versions. In many cases, the application source-code is no longer accessible requiring these applications to continue to run on outdated configurations. There is a huge demand of a framework which can recover functionally correct source-code components from such legacy software, so that such legacy systems can be ported to secure configurations.

Source-code analysis not sufficient: There are several scenarios where the source-code analysis is not sufficient. An executable code might demonstrate different behavior from the original source code. This phenomenon is popularly known as What-you-see-is-not-what-you-execute [20]. Modifications can happen to the source code during compilation (optimizations) or after the compilation process (bad code

injection). These modifications can significantly alter the program behavior. Consequently, the exact behavior of any program can only be uncovered by analyzing the executable code.

Moreover, several components of a typical software might be developed in multiple languages (Fortran, C and C++). The presence of different languages complicate the task of analyzing the source-code. In such scenarios, a consistent representation of the resulting executable code presents a more coherent analysis opportunity.

1.2 Advantages of executable analysis

Section 1.1 underscored the underlying importance of executable-level tools. In addition, executable level tools offers many advantages over standard compiler frameworks. Below, we discuss some of these advantages

- **End-user security enforcement.** Despite considerable research work on several computing hierarchies, low-level software vulnerabilities remain an important source of compromises and a perennial threat to system security. At the core, there exists a fundamental dichotomy in the capabilities and motivation of producers and consumers of software, vendors and end-users respectively. On the one hand, software producers are probably in the best position to prevent and mitigate such vulnerabilities: they have access to the source code. As a result, they can apply security mechanisms that offer high coverage and effectiveness at low overhead, because they are applied at the point where

the most semantic knowledge about the program and the code is available. On the other hand, it is software consumers that face the risk and bear the costs of compromise due to software vulnerabilities and are the most motivated to mitigate a newly discovered vulnerability. However, consumers often only have access to the program binary and configuration files. An executable-tool can bridge the gap between incentive/motivation and capabilities on the consumer side by enabling the end-users to retrofit custom security schemes into untrusted binaries, to prevent them from taking unauthorized actions.

- **Platform aware optimizations** Binaries compiled for wide distribution are often targeted for one particular ISA and are rarely optimized for a particular processor. Binary tools on an end-user platform can apply custom transformations to take advantage of platform-specific information like exact knowledge of the memory hierarchy or the precise version of multimedia instructions.
- **Whole-program analysis/optimizations.** Development toolchains typically employ separate compilation framework to minimize the compilation time. Hence, even though the compilers can theoretically do whole-program analyses, the applicability of such analysis is severely limited. In contrast, executable-level tools operate on the merged compilation units, allowing them to perform whole-program analyses on the compiled programs. Inter-procedural link-time analyses are usually far less precise than compile-time optimizations since they work on low-level object code without the benefit of the extensive IR features available in the compiler.

- **Economic feasibility.** An executable-level tool works for the code produced from any source language and by any compiler. Hence, it is more efficient to implement the transformations once in an executable-tool than repeatedly in each compiler. The high expense of repeated compiler implementation often cannot be supported by a small fraction of the demand¹.

1.3 Thesis Statement

In order to effectively operate under the above mentioned applications scenarios, an executable analysis system needs to perform similar to a source-code analysis framework. However the reality of executable-tools today has fallen far short of this desired vision. Existing binary frameworks [102, 107, 81, 33, 122, 66, 130, 149] are a little more than tools for peephole optimization and instrumentation.

It is conventional wisdom that static analysis of executables is a very difficult problem. There are several contributing factors towards the complexity of static analysis such as undecidable nature of static disassembly [82] and loss of semantic information during the compilation process [91]. These difficulties have resulted in a plethora of dynamic binary frameworks [102, 107, 81] and frameworks relying on metadata information [130, 95, 103, 149] to compensate for the loss of information during compilation and to circumvent the undecidable nature of disassembly. However, such metadata information is not present in commercial applications.

Our approach in this work is to enlarge the envelope of the types of program

¹This cost barrier to repeated implementation is a partial explanation of why, for example, there is a dearth of good commercial automatically parallelizing compilers, despite much progress in research prototypes in that area over three decades.

that can be handled by static analysis and rewriting. *Our main thesis is scalable static binary rewriting and analysis using compiler-level intermediate representation without relying on the presence of metadata information.* Below, we discuss the primary features of our thesis, followed by the assumptions under which we demonstrate the evidence of our thesis.

- **Static Framework:** There are several dynamic frameworks for analyzing executables [102, 107, 81, 33, 131]. Dynamic frameworks analyze a program while it is executing. Hence, the analysis time gets added to the application execution time. They have several limitations such as access to small portions of code at a time and huge overhead for advanced analyses. Without access to the complete code, it is extremely difficult to reason about the behavior of an application. Hence, our belief is that we can never attain our vision of reaching source-code analysis with dynamic framework. Hence, we focus our attention on static executable analysis frameworks.
- **Functionality:** Our executable framework recovers a functional representation from an executable, so that a correct executable can be obtained subsequently. This is essential for several applications such as debugging a malware or enforcing security schemes into applications.
- **Capability:** An executable framework must have no extra constraints in their representation as compared to source-code. The presence of such extra constraints in frameworks hitherto restricts the application of source-level research methods directly to executables.

- **Practicality:** Several frameworks assume the presence of metadata information such as debug information or relocation information which are not present in real world applications. A practical executable framework should not make such unreasonable assumptions.
- **Scalability:** An ideal executable framework must be able to scale to real world applications.

1.3.1 Assumptions behind this work

Correctly rewriting all binary programs is very challenging, hence our goal is to analyze and rewrite all compiled binaries. To this end, we declare a set of assumptions and define a variety of techniques to successfully handle the programs adhering to these set of assumptions. The methods proposed in this work rely on the following assumptions, which constitute our limitations. This work demonstrates that compiled code meets all these assumptions.

- **Disassembly assumptions:** The underlying disassembler employed in our framework derives possible addresses using the restriction that an indirect control transfer instruction requires an absolute address operand [135]. A compiled code is expected to adhere to this convention unless it has been generated to be position independent. Position-independent code (PIC) is typically generated only for standalone dynamically linked library code, which we currently cannot rewrite. Application code (with statically linked libraries or calls to external DLLs) is handled without any restrictions. However, other researchers in our group are looking

to overcome this assumption by rewriting PIC code as well.

- **Obfuscated Code:** In order to protect intellectual property, some commercial programs employ obfuscation mechanisms to enhance the resistance against reverse engineering tools. A variety of obfuscation mechanisms have been proposed which make it harder to precisely construct a control flow graph. This includes excessive use of indirect control transfers and usage of non-standard procedure transfers without using the `call/return` mechanism. Debray et al [99] have proposed more advanced obfuscation mechanisms such as branch functions to thwart static disassembly. This work assumes that executables do not employ such kind of obfuscations mechanisms and recursive disassembly. With these assumptions, the techniques presented by Smithson and Barua [135] and Wazeer et al [63], are successful in achieving 100% code coverage.
- **Memory assumptions:** Similar to most executable analysis frameworks [20, 22, 48, 130], our techniques assume that executables follow the *standard compilation model* where each procedure optionally maintains a local frame, which grows in only one direction and each variable resides at a fixed offset in its corresponding region. We also assume that in x86 programs, a particular register `esp` refers to the top of memory stack. This assumption is expected to hold in all practical scenarios since x86 ISA inherently makes this assumption. For example, `call` instruction moves `eip` to `esp` and `return` decrements `esp`. Moreover, interrupt handler codes that are part of an operating system, and can be called during an application program at any time, inherently assume a stack that follows these

restrictions. Such handler codes typically allocate their own data on top of the stack growing in one known direction pointed to by `esp`. Such mechanisms would not work in programs without such a stack. An assembly code not adhering to this convention would be extremely hard to write.

- **Memory Consistency:** Our framework mimics the assumptions behind all standard software transformation tools with regards to memory consistency. A majority of compilers (`gcc`, LLVM, Visual Studio) and popular binary frameworks like PLTO [130], DynamoRIO [33], PIN [102], iSpike [103], Diablo [149] reorder code without taking memory consistency into account. Since synchronization is highly multiprocessor specific, most programmers are expected to write synchronized programs using standard synchronization libraries [104]. The presence of synchronization primitives legalizes the applications of all software optimizations. Recently, the research community is exploring the possibility of preserving memory consistency in software transformation tools [104]. The key idea is to selectively invalidate the transformations for possibly shared memory locations. In current implementation, our framework can preserve consistency by declaring all possibly shared memory regions as *volatile* in the IR.
- **Self Modifying code:** Like most static binary tools, we do not handle self modifying code. Various tools [156] statically detect the presence of self-modifying code in a program. Such a tool can be integrated in our front-end to warn the user and to discontinue further operation.

In the next section, we discuss some of the limitations of existing executable frameworks and discuss the contributions made by this work in eliminating these limitations.

1.4 Contribution of this dissertation

As mentioned above, our main thesis is scalable static binary rewriting and analysis using compiler-level intermediate representation without relying on the presence of metadata information. Figure 1.1 depicts the overall contribution of this dissertation. Below we briefly discuss the individual contributions.

1.4.1 Representation

As part of this work, we have tried to resolve what we believe is a fundamental aberration - in spite of a significant overlap in the overall goals of several source-code methods and executables-level techniques, several sophisticated transformations that are well-understood and implemented in source-level infrastructures have yet to become available in executable frameworks. Many of the executable-level tools suggest new techniques for performing elementary source-level tasks. For example, PLTO [130], a link-time optimizer, proposes a custom alias analysis technique to implement a simple transformation like constant propagation in executables. Similarly, several techniques for detecting security vulnerabilities in source-code [154, 37] remain outside the realm of current executable-level frameworks.

It is a well known fact that a standalone executable without any meta data

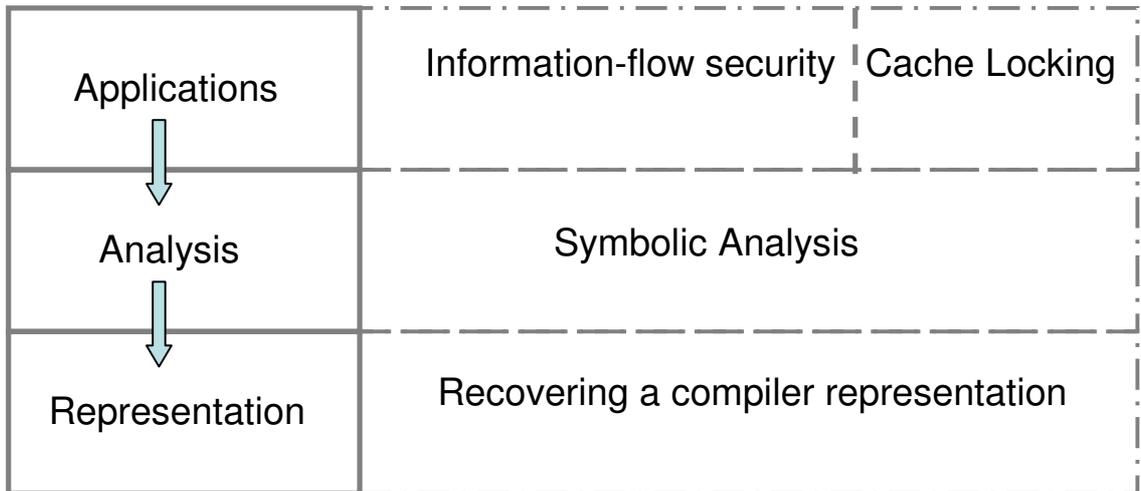


Figure 1.1: *Contributions of the dissertation*

is less amenable to analysis than the source-code. Nonetheless, we believe that one of the prime reasons behind the underlying aberration is that current executable frameworks define their own intermediate representations (IR) which are significantly more constrained than an IR used in a compiler. Intermediate representations used in existing binary frameworks lack high level features such as abstract stack, variables, and symbols and are even machine dependent in some cases. This severely limits the application of well-understood compiler transformations to binaries and necessitates new research to make them applicable.

In order to achieve our aim of a *capable* and a *functional* executable framework, we present techniques to convert the binaries to the same high-level intermediate representation that compilers use. We present techniques to segment the flat address space in an executable containing undifferentiated blocks of memory. We demonstrate the inadequacy of existing variable identification methods for their promotion to symbols and present our methods for symbol promotion. We also present meth-

ods to convert the physically addressed stack in an executable (with a stack pointer) to an abstract stack (without a stack pointer). The proposed methods are *practical* since they do not employ symbolic, relocation, or debug information which are usually absent in deployed executables.

The compiler IR is then employed for three distinct applications: binary rewriting using the compiler’s binary back-end, vulnerability detection using existing source-level symbolic execution tools, and source-code recovery using the compiler’s C backend. Our techniques enable complex high-level transformations not possible in existing binary systems, address a major challenge of input-derived memory addresses in symbolic execution and are the first to enable recovery of a fully functional source-code.

1.4.2 Analysis

The effectiveness of any tool is governed by the effectiveness of its underlying analysis frameworks. A source level program analysis framework typically employs multiple static analyses for analyzing and optimizing the programs. Symbolic analysis is an important static analysis method where the values of program variables and expressions are represented through symbolic expressions in an abstract domain. Symbolic analysis has been shown to improve the efficacy of various program analyses such as global value numbering and dependence analysis.

However, such source-level symbolic analysis frameworks have limited effectiveness in the executable domain since executables typically lack higher-level semantics

like variable and structures and mainly contain memory locations instead of explicit program variables. The IR should have a precise memory abstraction for an analysis to effectively reason about memory operations.

Our techniques of recovering a compiler-level intermediate representation address this limitation by recovering several higher-level semantics information from executables. Below, we propose two techniques to handle the scenarios when such semantics cannot be recovered.

First, executable specific artifacts such as indirect control transfers complicate the task of recovering a precise memory abstraction while maintaining the *functionality* in IR. The lack of a precise memory abstraction constrain the efficacy of several executable analyses. We propose a hybrid static-dynamic mechanism for recovering a precise and correct stack memory model in executables in presence of executable-specific artifacts.

Next, the enhanced memory model is employed to define a novel symbolic analysis framework for executables that can perform the same types of program analysis as source-level tools. Frameworks hitherto fail to simultaneously maintain the properties of correct representation and precise memory model and ignore memory-allocated variables while defining symbolic analysis mechanisms. The proposed symbolic analysis framework for executables adapts source-level symbolic analysis framework to perform well even in the absence of higher level semantics. We exemplify that our framework is robust, efficient and it significantly improves the performance of various traditional analyses such as global value numbering (GVN), alias analysis and dependence analysis for executables. Such a powerful symbolic

analysis framework can improve the effectiveness of any binary analysis tool where it is employed.

1.4.3 Applications

The underlying representation and analysis framework is employed for two separate applications. First, the framework is extended to define a novel static analysis framework, *DemandFlow*, for identifying information flow security violations in executables. Unlike existing static vulnerability detection methods for executables, DemandFlow analyzes memory locations in addition to symbols, thus improving the precision of the analysis. DemandFlow proposes a novel demand-driven mechanism to identify and precisely analyze only those program locations and memory accesses which are relevant to a vulnerability, thus enhancing scalability. Since DemandFlow uses static analysis, it does not incur a runtime performance overhead. In contrast to other similar analyses, DemandFlow also does not require source code.

Next, the framework is extended to implement a platform-specific optimization for embedded processors. Various different approaches have been suggested to enable software involvement in the management of the on-chip memory. Several embedded systems such as Intel's XScale and ARM's latest Cortex processors provide the facility of locking one or more lines in the cache - this feature is called cache locking. In spite of the presence of cache locking mechanism in modern processors, there are no methods in the literature to employ cache locking for improving cache performance. We devise the first method in literature employing instruction cache

locking as a mechanism for improving the average-case run-time of general embedded applications. We demonstrate that the optimal solution for instruction cache locking can be obtained in polynomial time. However, the nature of cache locking in existing hardware renders such optimal solutions impractical. Instead, we propose two practical heuristics based approaches to achieve cache locking.

We reckon that portability is one huge issue for successful implementation of cache locking inside a compiler. Cache locking inside a compiler yields executables that are tied to a particular cache size known at compile-time. The executables are not portable to other cache sizes. Cache sizes often increases among successive processor generations of the same instruction set, as predicted by Moore's law. This is particularly troublesome when the same code is downloaded to each node in long-lived networks of embedded systems, each with possibly different memory sizes. Modern processors employ diverse memory hierarchy with cache sizes varying in sizes and correspondingly varying in amount of locking involved.

Since our scheme is implemented inside a binary framework, it successfully addresses the portability concern by enabling the implementation of cache locking at the time of deployment when all the details of the memory hierarchy are available. This work proposes a next-generation cache locking aware memory manager where the memory manager resides entirely in the install-time system, and NOT the compiler. In such a scenario, low-level memory management will be a service provided by the install-time system in concert with the hardware, just like virtual memory. Thus the memory management will be transparent to the software toolchain. To our knowledge, there are no successful install-time-only systems for software involved

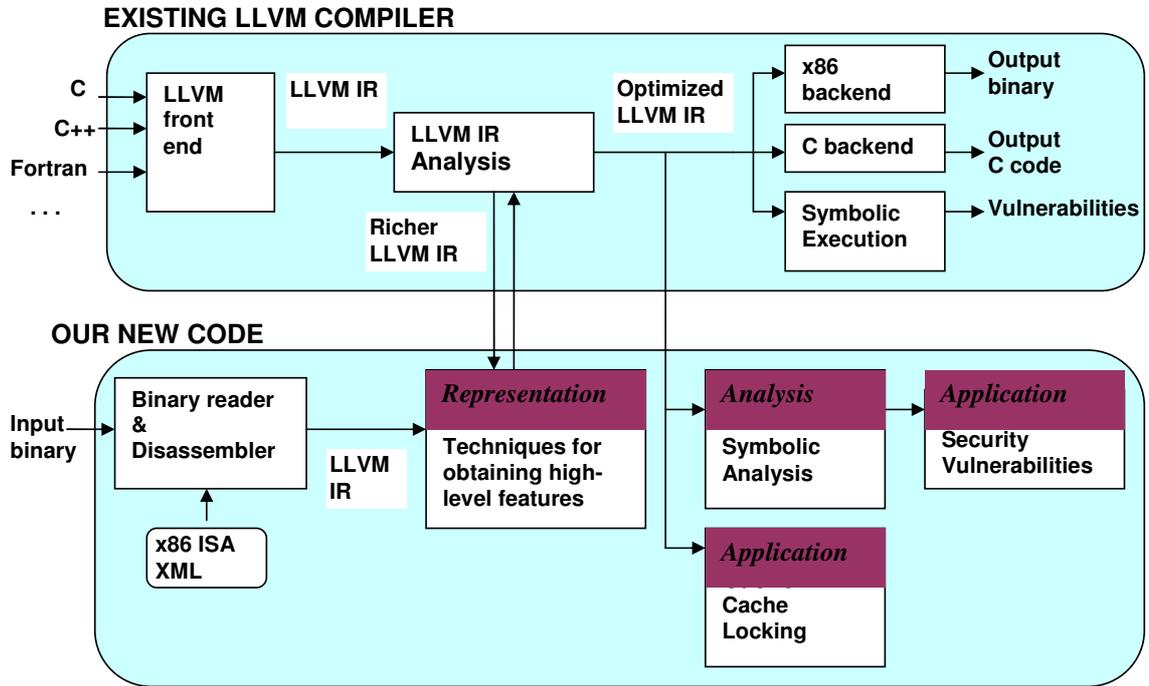


Figure 1.2: *SecondWrite* system highlighted with the contributions of this dissertation

cache management in the literature.

1.5 SecondWrite

We have achieved all the above advantages by implementing our techniques in a binary rewriter called SecondWrite [10], which employs the widely used open-source Low-level Virtual Machine (LLVM) [96] compiler IR to represent the code. Fig 1.2 shows the flow of SecondWrite system, highlighting the techniques proposed in this dissertation.

SecondWrite’s custom binary reader and decompiler modules parse the input binary and produce a functionally equivalent LLVM IR code [135]. The disassembler also implements several additional techniques [63] to recover procedure boundaries

and inserts additional checks that are essential for the IR to be functional in case of inaccurate recovered boundaries. Our techniques for obtaining a high-level representation convert this initial LLVM IR to a richer LLVM representation, containing abstract stack and symbols. The symbolic analysis framework proposed in this dissertation is built over the LLVM representation recovered in the previous step. Thereafter, the recovered representation is employed for two distinct applications. First, the symbolic analysis framework is extended to define a scalable and precise framework for uncovering information-flow vulnerabilities in executables. Second, the recovered representation is employed in implementing a cache locking mechanism for embedded processors.

SecondWrite is a highly collaborative research effort and was developed as part of this dissertation in close efforts with Matthew Smithson, Khaled Elwazeer and Aparna Kotha. Matthew Smithson and Khaled Elwazeer are primarily responsible for techniques behind the disassembler module, Khaled and Aparna have major contributions in implementing techniques for obtaining a richer LLVM representation containing semantic variable types and adequate representation of floating point x86 registers. Aparna also developed methods to parallelize binary executables and implemented condition handling in SecondWrite.

Converting binaries to compiler-level IR acts as a great baseline for applying binary-to-binary optimizations, as described below:

- **Ability to do any code transformation** Using compiler IR enables every compiler transformation to run without binary-specific customization on any

binary.

- **Ability to do effective compiler analysis and optimization** Using compiler IR with presence of variables and symbols allows dataflow analysis to become much more effective.
- **Reuse compiler research** A compiler IR allows rewriter to leverage a substantial body of work on source-level analysis by enabling the application of existing compiler level research to binary rewriters without any modifications.
- **Binary to source conversion** Existing compiler backends can be used to convert the IR obtained from binary to source languages like C for better code understanding of binaries with no source.
- **Reuse passes from mature compilers** Sharing the IR with a mature compiler allows the binary rewriter to leverage the full set of compiler passes built up over decades by hundreds of developers.

1.6 Organization of the Dissertation

In this dissertation, we demonstrate that a static binary framework based on a compiler IR enables applications not possible in any existing tool and our results establish the feasibility of this approach for most pragmatic scenarios. We do not claim that we have fully solved all the issues; statically handling every program in the world may still be an elusive goal. However, the resulting experience of expanding the static envelope as much as possible is a hugely valuable contribution

to the community.

The dissertation is organized as follows.

- Chapter 2 highlights our contributions in light of several existing executable analysis frameworks.
- Chapter 3 presents the techniques for recovering compiler-level intermediate representation from executables. It constitutes our methods for symbol promotion and for converting the physically addressed stack in an executable to an abstract stack. Our system is the first to demonstrate that a compiler intermediate representation can be successfully employed in a static binary framework.
- Chapter 4 discusses our symbolic analysis framework for executables. The proposed symbolic analysis framework enhances the efficacy of several transformations on executables such as value numbering by 40% on average.
- Chapter 5 discusses the techniques to improve memory abstraction in presence of executable specific artifacts such as indirect control transfers. Our techniques improve the precision of memory abstraction by 15% on average in programs containing such artifacts.
- Chapter 6 extends our underlying representation and analysis framework for uncovering information flow vulnerabilities in executables. Our tool uncovers six previously unknown vulnerabilities in popular internet and relay chat programs at a low false positive rate of 79%.

- Chapter 7 addresses the problem of cache locking. We present the first method in the literature for cache locking that is able to reduce the average-case runtime of a program. Our mechanism results in 32% improvement in execution time in memory constrained embedded applications.
- Finally, Chapter 8 postulates future research directions and discusses the conclusions of this dissertation.

Chapter 2: **Related Work**

In this chapter, we discuss the related work in the broad field of executable analysis and rewriting frameworks. The techniques related to individual techniques are discussed separately within each chapter. Fig 2.1 summarizes the comparison of our framework with existing executable frameworks, in light of individual features introduced in Section 1.3.

2.1 Binary rewriting

Binary rewriting research is being carried out in two directions: static and dynamic. Dynamic binary rewriters rewrite the binary during its execution. Examples are PIN [102], BIRD [107], DynInst [81], DynamoRIO [33] and Valgrind [131]. None of the dynamic binary rewriters we found employ an IR of an existing compiler. This is not surprising since dynamic rewriters construct their internal representation at run-time, and hence they would not have the time to construct a compiler IR. Dynamic rewriters are hobbled since they do not have enough time to perform complex compiler transformations either; they have been primarily used for code instrumentation and simple security checks in the past. We do not discuss dynamic rewriters

Property	Rewrites correctly	High IR	Works without Metadata	Scalable
ATOM (Link time)	✓	X	X	✓
PLTO (Link time)	✓	X	X	✓
Spike (Link time)	✓	X	X	✓
UQBT	✓	X	X	✓
IDA Pro / Hex Rays	X	✓	✓	✓
Jakstab	X	X	✓	X
BAP (TIE)	X	✓	✓	X
CodeSurfer/X86	X	✓	✓	X
SecondWrite	✓	✓	✓	✓

Figure 2.1: *Comparing SecondWrite with other executable tools*

further since our methods are primarily directed at static binary frameworks.

Existing static binary rewriters related to our approach include Etch [122], ATOM [66], PLTO [130], Diablo [149], Spike [103] and UQBT [48]. All these rewriters define their own low-level custom IR as opposed to using a compiler IR. These IRs are devoid of features such as abstract frames, symbols and maintain memory as a flat address space; the limitations of which have already been discussed in

Chapter 1. Diablo defines an augmented whole program control-flow-graph-based intermediate representation with program registers as globals and memory as a black box. It does not attempt to obtain high-level information such as function prototypes and is geared mainly towards optimizations like code compaction. Taking memory as a black box limits its applicability to architectures such as x86 which contain very small number of registers. ATOM defines a symbolic RTL-based intermediate format with infinite registers but does not do any attempt of analyzing or modifying the stack layout. It is mainly targeted towards RISC architectures like Compaq Alpha. PLTO employs a whole program CFG based IR and implements stack analysis to determine the use-kill depths of each function [58]. However this information is not used for converting it into high-level IR; rather it is used only for low-level custom optimizations like load/store forwarding. Etch does not explicitly build an intermediate representation and allows user to add new tools to analyze binaries. The primary goal of Etch appears to be instrumentation and has only been shown to be applicable for simple optimizations like profile-guided code layout. Some post-link time optimizers like Spike [103] promote memory locations to symbols employing the symbol table information in the object files. However, deployed binaries do not contain symbol information, rendering such solutions to be impractical for executables.

UQBT [48] is a binary translation framework which defines its own custom intermediate format as opposed to using an existing compiler's IR; hence it loses out on the advanced set of optimizations implemented in an already existing mature compiler infrastructure. The IR involved is high level involving procedure prototype

abstraction but the conversion to IR relies on user-provided information about the number of parameters and their locations, instead of determining that information automatically from a binary like we do. This severely limits the applicability of UBQT since only the developers have access to that information, and moreover, the translation process to an intermediate form is no longer automatic.

Virtual machines [6] implement stack-walking techniques to determine the calling context by simply iterating over the list of frame pointers maintained as metadata in the dynamic framework; making it orthogonal to our mechanism which statically inserts run-time checks in the IR.

2.2 Binary Analysis/Intermediate Representation recovery

There are several executable analysis tools such as BAP [35], BitBlaze [137], Phoenix [114] and others which recover an IR from an executable for further analysis. However, these tools have several limitations. All these tools define their own custom IR without the features of abstract stack and symbol promotion, facing limitations similar to tools like Diablo [149] discussed above. Phoenix [114] recovers a register transfer language (RTL) resembling architecture neutral assembly, which does not expose the semantics of several complicated instructions. Further, Phoenix and several other tools [95] require debugging information, which is usually absent in deployed executables.

Various executable frameworks ease the specification of semantics of native instructions [141] which is orthogonal to our task of recovering intermediate repre-

sentation. Tools like Jakstab [89] address control flow challenges in executables by resolving indirect branches using multiple rounds of disassembly interleaved with dataflow analysis. However, they do not recover any high level information from executables and have been shown to scale to programs of a limited size.

There are some frameworks which recover LLVM IR from executables. S2E [47] and RevNIC [45] present a method for dynamically translating x86 to LLVM using QEMU. Unlike our approach, these methods convert blocks of code to LLVM on the fly which limits the application of LLVM analyses to only one block at a time. RevNIC [45] recovers an IR by merging the translated blocks, but the recovered IR is incomplete and is only valid for current execution; consequently, various whole program analyses will provide incomplete information. RevGen [46] includes a static disassembler to recover an IR for entire binary. However, the translated code retains all the assumptions of the original binary about the stack layout. They do not provide any methods for obtaining an abstract stack or promoting memory locations to symbols, which are essential for the application of several source-level analyses.

King et al. [90] provide a comprehensive survey of several executable analysis tools. Balakrishnan et al. [20, 22] present Value Set Analysis for analyzing memory accesses and extracting high level information like variables and their types. As we will discuss in detail in Chapter 3, analyzing variables does not guarantee promotion to symbols in IR. Zhang et al. [164] present techniques for recovering parameters and return values from executables but they do not consider the scenarios where the information cannot be derived. As mentioned before, such best effort solutions are good for executable analysis but do not certify the reliable behavior once these

analyses fail.

Jianjun et al. [97] promote stack variables to registers dynamically, relying on hardware mechanism for memory disambiguation. In contrast, we provide techniques for symbol promotion in a static framework without any hardware support.

2.3 Industrial Tools

There are three popular industrial-level tools for analyzing executables - HexRays [80], CodeSurfer/x86 [19] and Veracode [13].

The Hex-Rays decompiler [80] (the sister product to the IDA Pro disassembler) is a commercially-available decompiler. Unfortunately, the product and its research are proprietary, and its inner workings are closely guarded trade secrets. Thus they are not available for others to replicate. However, two drawbacks are apparent from their website. First, they acknowledge is that their output is not 100% reliable (perhaps because of the inherent uncertainties of disassembly), whereas our techniques always generates functional output. Second, they only support binaries compiled from C/C++ using standard compilers. We conjecture that these could be because they make language and compiler-specific assumptions. This severely limits their applicability in practical scenarios.

CodeSurfer/x86 [19] is built on the techniques suggested by Balakrishnan et al. [20, 22]. As mentioned before, such best effort solutions are good for executable analysis but do not certify the behavior once these analyses fail. As opposed to our techniques, it fails to maintain the functionality of the recovered intermediate

representation.

Veracode [13] uncovers vulnerabilities in executables. To the best of our knowledge, the techniques used by Veracode are proprietary and have not been published anywhere. Hence, the underlying techniques cannot be compared. Further, unlike our techniques, Veracode requires the presence of debug information, which is not present in deployed executables.

Chapter 3: **Decompilation to compiler level intermediate representation**

3.1 Introduction

We have identified the two tasks below as key for translating binaries to compiler IR. We illustrate the advantages of these two methods through the source-code recovered from a binary corresponding to the example code in Fig 3.1(a).

- **Deconstruction of physical stack frames** A source program has an abstract stack representation where the local variables are assumed to be present on the stack but their precise stack layout is not specified. In contrast, an executable has a fixed (but not explicitly specified) physical stack layout, which is used for allocating local variables as well as for passing the arguments between procedures.

To recreate a compiler IR, the physical stack must be deconstructed to individual abstract frames, one per procedure. Since the relative layout of these frames might change in the rewritten binary, the correct representation requires *all* the arguments (interprocedural accesses through stack pointer) to be recognized and translated to symbols in the IR.

<pre>int main(){ int z; z = foo(10,20); return z; } int foo(int a, int b) { int temp3,temp1; temp1 = a+b; if(a>40){ temp3 = temp1 + 10; } else { temp3 = temp1 - 10; } return temp3; } (a) Original C Code</pre>	<pre>//Global Stack Pointer int* llvm_ESP; char *main(){ llvm_ESP = llvm_ESP-2; //Local Allocation llvm_ESP[1] = 20; //Outgoing argument llvm_ESP[0] = 10; int llvm_tmp_3 = rewritten_foo(); return llvm_tmp3; } int rewritten_foo() { int* llvm_EBP = llvm_ESP; //Local Frame Pointer llvm_ESP = llvm_ESP-10; //Local Allocation int tmpIn1 = llvm_EBP[0]; //Incoming Arg int tmpIn2; = llvm_EBP[1]; int llvm_tmp2 = tmpIn1+tmpIn2; llvm_ESP[2] = llvm_tmp2; int llvm_tmpIn3 = llvm_EBP[0]; if (llvm_tmpIn3 > 40){ int llvm_tmp5 = llvm_ESP[2]; llvm_ESP[5] = llvm_tmp5 + 10; } else { int llvm_tmp7 = llvm_ESP[2]; llvm_ESP[5] = llvm_tmp7 - 10; } int llvm_tmp11 = llvm_ESP[5]; return llvm_tmp11; } (b) Recovered C Code with physical stack</pre>	<pre>char *main() { int llvm_ESP2[10]; llvm_ESP2[1] = 20; llvm_ESP2[2] = 10; int llvm_tmp1 = llvm_ESP2[1]; int llvm_tmp2 = llvm_ESP2[2]; int llvm_tmp_3 = rewritten_foo(llvm_tmp2, llvm_tmp1); return llvm_tmp3; } int rewritten_foo(int llvmArg1, int llvm_Arg2) { int llvm_ESP1[10]; int llvm_tmp = llvm_Arg1+llvm_Arg2; llvm_ESP1[2] = llvm_tmp2; if (llvm_Arg1 > 40) { int llvm_tmp5 = llvm_ESP1[2]; llvm_ESP1[5] = llvm_tmp5 + 10; } else { int llvm_tmp7 = llvm_ESP1[2]; llvm_ESP1[5] = llvm_tmp7 - 10; } int llvm_tmp11 = llvm_ESP1[5]; return llvm_tmp11; } (c) Recovered C Code with abstract stack</pre>
<pre>char *main(){ int llvm_tmp3; llvm_tmp_3 = rewritten_foo(10,20); return llvm_tmp3; } int rewritten_foo(int llvm_Arg1, int llvm_Arg2){ int llvm_tmp4; int llvm_tmp2 = llvm_Arg1 + llvm_Arg2; if (llvm_Arg1 > 40){ llvm_tmp4 = llvm_tmp2 + 10; } else { llvm_tmp4 = llvm_tmp2 - 10; } return llvm_tmp4; } (d) Recovered C Code with abstract stack and symbol promotion</pre>		

Figure 3.1: *Source-code example. Variable names and types in the source-code recovered by LLVM C-backend have been modified for readability.*

Unfortunately, guaranteeing the static discovery of all the arguments is impossible. Some indirect memory references with run-time-computed addresses might make it impossible for an analysis to statically assign them to a fixed stack location, resulting in undiscovered interprocedural accesses. Existing frameworks circumvent this problem by preserving the monolithic unmodified stack in the IR, resulting in a low-level IR where no local variables can be added or deleted.

Some executable tools analyze statically determinable stack accesses to recognize *most* arguments [20], aiding limited code understanding. However, the lack of guaranteed discovery of *all* the arguments renders such best-effort techniques insufficient for obtaining a functional IR. Fig 3.2 shows an example procedure where

<pre>foo(int a, int b) { int *p, *q; p = &a; ... *q = ...; ... = b; }</pre>	<p>Stack q: edx p: esp + 8</p> <p>allocations a: esp + 20 b: esp + 24</p> <pre>foo: 1 subl \$16, %esp // Allocate 16-byte stack frame 2 lea 20(%esp), 8(%esp) // Put &a(esp+20) into p(esp+8) 3 store ..., (%edx) // Store to MEM[q] 4 load 8(%esp), %ecx // Temp ecx ← p (same as &a) 5 load 4(%ecx) // Load "b" by using the fact that &b = &a + 4 = ecx + 4</pre>
Source Code	Pseudo Assembly Code

Figure 3.2: A small source-code example and its pseudo-assembly code, showing the limitation of existing methods for detecting arguments.

the first argument **a** can be recognized statically while the second argument **b** is not statically discoverable. In the assembly-code, $\&a$ ($\text{esp}+20$) is stored to the memory location for **p** ($\text{esp}+8$) (Line 2), which is loaded later to temporary **ecx** (Line 4). The source compiler exploited the layout information ($\&a+4=\&b$) to load **b** by incrementing **p** ($\&a$) by 4 (Line 5). This is safe since the compiler was able to determine that **p** does not alias **q**. However, the executable framework may not be able to establish this relation, since alias analysis in executables is less precise. Hence, it has to conservatively assume that $*q$ reference (Line 3) could modify **p** which contained the pointer to **a**. Consequently, the source address at Line 5 is no longer known and argument **b** is not recognized.

Our analysis in Section 3.3 defines a source-level stack model and checks if the executable conforms to this model. If the model is verified for a procedure, the analysis discovers the arguments statically when possible, but when not possible, embeds run-time checks in IR to maintain the correctness of interprocedural dataflow. Otherwise, stack abstraction is discontinued only in that procedure.

Fig 3.1(c) demonstrates the impact of abstract stack on the recovered source-

<pre> main() { int A[10], i, x; x = read-from-file(); for (i = 0; i < x; i++) { A[i] = 10; } } </pre>	<pre> main: 1 subl \$48, %esp 2 %ebx = read_from_file 3 mov %ebx, 44(%esp) //Initializing x 4 movl \$0, 40(%esp) //Initializing i 5 jmp L2 // jump to condition check L3: 6 movl 40(%esp), %eax //load i 7 movl \$10, (%esp,%eax,4) //Reference A[i] 8 addl \$1, 40(%esp) //Increment i L2: 9 cmpl 40(%esp), 44(%esp) //compare x and i 10 jl L3 </pre>
--	--

Figure 3.3: An example showing that variable identification and symbol promotion are different.

code. Fig 3.1(b) employs a global pointer `llvm_ESP`, corresponding to the physical stack frame in the input binary, for interprocedural communication as well as for representing local allocations in each procedure. However, in Fig 3.1(c), the stack pointer disappears; instead, local allocations appear as separate local arrays `llvm_ESP1` and `llvm_ESP2` and arguments are represented explicitly.

- **Symbol promotion** Another key challenge we solve is *symbol promotion*, which is the process of safely translating a memory location (or a range of locations) to a symbol in the recovered IR. Existing frameworks do not promote symbols; instead they retain memory locations in their IR [130, 149, 102, 122]. Some post-link time optimizers like Ispike [103] promote memory locations to symbols employing the symbol table information in the object files. However, deployed binaries do not contain symbol information, rendering such solutions unsuitable for our framework.

At first glance, it may seem that the well-known methods for variable identification in executables, such as IDAPro [84] and Divine [22], can be used for symbol

promotion. However, this is not the case. The presence of potentially aliasing memory references is a key hindrance to the valid promotion of these identified variables to symbols.

IDAPro characterizes statically determinable stack offsets in the program as local variables while Divine divides the stack memory region into abstract locations by analyzing indirect memory accesses instructions as well.

Fig 3.3 illustrates the key limitations of both these methods. When the code is compiled, we obtain a stack frame for `main()` of size 48 bytes (10×4 bytes for `A[]`, and $4 \times 2 = 8$ bytes for `i` and `x`). The accesses to variables `i` and `x` appear as direct memory references (Lines 3,4,6,9) while the array `A` is accessed using an indirect memory reference (Line 7). Both Divine and IDAPro identify memory locations `esp+44(x)` and `esp+40(i)` as variables based on the direct references. Since the upper bound for the indirect reference `A[i]` is statically indeterminable, even Divine does not generate any useful information about this access. Hence, it creates three abstract locations — two scalars of 4 bytes each, and a leftover range of 40 bytes.

Despite dividing stack memory region into three abstract locations, none of them can be promoted to symbols. It is impossible to statically prove from an executable that the indirect reference at Line 7 does not alias with references to `i` or `x`. Hence, the promotion of memory locations corresponding to `i` and `x` to symbols would be unsafe since it leads to potentially inconsistent dataflow for underlying memory locations. (Source-level alias analyses often assume that any `A[x]` will access `A[]` within its size. However, such size information is not present in a stripped executable.)

Since identification is inadequate for promotion, we have devised a new algorithm to safely promote a set of memory locations to symbols. It computes a set of non-overlapping promotion lifetimes for each memory location taking into consideration the impact of aliasing memory accesses. Our method is oblivious to the underlying method employed for identifying these locations. The locations can be identified by IDAPro, Divine or through a similar method we use.

Fig 3.1(d) shows the improvement in source-code recovery from symbol promotion, illustrating the replacement of all access to local array `11vm_ESP1` and `11vm_ESP2` in procedures `foo` and `main` respectively by local symbols. As evident, this greatly simplifies the IR and the source-code.

3.1.1 Benefits of abstract stack and symbols

The presence of abstract stack and symbols has the following advantages:

- Improved dataflow analysis since standard dataflow analyses only track symbols and not memory locations.
- Improved readability of the recovered source-code.
- The ability to employ source-level transformations without any changes. Advanced transformations like compiler-level parallelization [148, 165] add new local variables as barriers and rely on the recognition of induction variables. Several compile-time security mechanisms like StackGuard [55] and ProPolice [65] modify stack layout by placing a *canary* (a memory location) on the stack or by allocating local buffers above other local variables. These methods

can be implemented only if the framework supports stack modification and symbol promotion.

→ Efficient reasoning about symbolic memory in case of symbolic execution, as discussed next.

3.1.1.1 Symbolic Execution

Symbolic execution, e.g. [39], is a well-known technique for automatically detecting bugs and vulnerabilities in a program. Among various challenges facing symbolic execution, handling symbolic memory addresses (addresses derived from user-input) is an important one. There are two primary approaches for handling symbolic memory. Previous symbolic executors for executables [137] make simplifying and unsound assumptions by concretizing the symbolic memory reference to a fixed memory location. On the other hand, popular source-level tools [39, 38] employ logical constraint solvers to reason about possible locations referenced by a symbolic memory operation. Even though the expressions involving symbolic memory become more sophisticated, these tools outperform the former approaches in terms of path exploration and bug detection [42].

The presence of a physical stack and the lack of symbols in an executable pose a difficult challenge in efficiently extending the logical solver based approach for representing symbolic memory in executables. The most straightforward representation of the memory would be a flat byte array. Unfortunately, the constraint solvers employed in existing source-level symbolic execution tools would almost never be able

	x: esp+48 y: esp + 44	esp+48: symX esp+44: symY
<pre>foo() { int A[10], x,y; x = read-from-file(); y= read-from-file(); if(x<10) { A[x] = 30; } if(y>20) { return; } }</pre>	<p>L0:</p> <pre>1 FOO= alloca i32,48 2 ebx1 = read_from_file() 3 store ebx1, 48(FOO) //store x 4 ebx2 = read_from_file() 5 store ebx2, 44(FOO) //store y 6 ebx3 = load 48(FOO) //load x 7 if(ebx3>=10), jmp L2:</pre> <p>L1:</p> <pre>8 store \$30, FOO[4*ebx3]</pre> <p>L2:</p> <pre>9 eax = load 44(FOO) //load y 10 if(eax<=20) jmp L3 return</pre> <p>L3:.....</p>	<p>L0:</p> <pre>1 FOO= alloca i32,48 2 ebx1 = read_from_file() 3 symX=ebx1 4 ebx2 = read_from_file() 5 symY = ebx2 6 ebx3= symX 7 if(ebx3>=10), jmp L2:</pre> <p>L1:</p> <pre>8 store \$30, FOO[4*ebx3]</pre> <p>L2:</p> <pre>9 eax = symY 10 if(eax<=20) jmp L3 return</pre> <p>L3:.....</p>
a) Original Code	b) IR without symbol promotion	c) IR with symbol promotion

Figure 3.4: An example showing the simplification in symbolic execution constraints with symbol promotion.

to solve the resulting constraints [38].

<p>Constraints:</p> <pre>A1=write(FOO,48,ebx1) A2= write(A1,44,ebx2) read(A2,48)<10 A3 = write(A2, 4*read(A2,48),30) Solve: read(A3,44) <= 20</pre>
--

Figure 3.5: Constraints for Fig 3.4(b).

The segmented memory representation in our framework, obtained by abstract stack and symbol promotion, improves the efficiency of such constraint solvers by enabling them to only consider the constraints related to the segments referenced by the current memory address expression and ignore the remaining segments.

Fig 3.4 illustrates this case. Fig 3.4(a) contains a symbolic memory store to array A. Fig 3.4(b) and Fig 3.4(c) show the pseudo IR obtained from an executable corresponding to Fig 3.4(a), without and with the application of symbol promotion. Fig 3.5 shows the constraints and query generated at Line 10 while symbolically executing the path L0→L1→L2 in Fig 3.4(b). Here, `read(A,i)` returns the value at index `i` in array A and `write(A,j,v)` returns a new array with same value as A at all indices except `j`, where it has value `v`.

However, in Fig 3.4(c), symbol promotion has segmented the array F00 in different segments and references to variables `x` and `y` do not refer the segment F00. Hence, the solver only needs to solve the following simplified query:

$$\text{Solve : } \text{sym}Y \leq 20$$

This example only shows the simplification of constraints with symbol promotion. The presence of an abstract stack also results in a similar simplification of constraints by segmenting the memory space within each procedure.

3.2 Overview of the framework

Fig 3.6 presents an overview of the SecondWrite framework. The frontend module, consisting of a disassembler and a custom reader module, processes the individual instructions in an input executable and generates an initial LLVM IR. The framework implements several techniques [62] for recognizing arguments passed through registers and for handling floating point registers. This initial IR is devoid of the

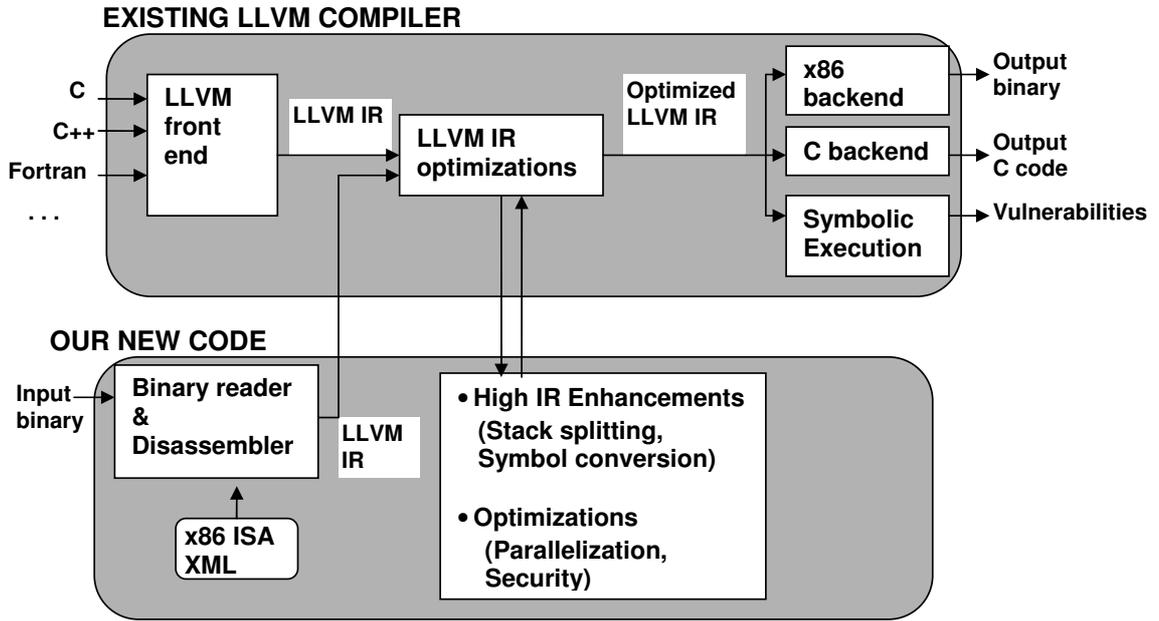


Figure 3.6: *SecondWrite* system.

desired features like abstract stack frame and symbols. This initial IR is analyzed to obtain an enhanced IR which has all the information and features mentioned previously.

SecondWrite has been already been employed for several applications such as automatic parallelization [93] and security enforcements [111]. As discussed in Section 3.1, the features of abstract stack and symbols are critical for an efficient implementation of these applications.

3.2.1 Disassembler Module

The disassembler module implements several mechanisms, as proposed by Smithson and Barua [135], to address code discovery problems and to handle indirect control transfers. Here, we briefly summarize these mechanisms.

A key challenge in executable frameworks is discovering which portions of the

code section in an input executable are definitely code. Smithson and Barua [135] proposed *speculative disassembly*, coupled with *binary characterization*, to efficiently address this problem. SecondWrite speculatively disassembles the unknown portions of the code segments as if they are code. However, it also retains the unchanged code segments in the IR to guarantee the correctness of data references in case the disassembled region was actually data.

SecondWrite employs *binary characterization* to limit such unknown portions of code. It leverages the restriction that an indirect control transfer instruction (CTI) requires an absolute address operand, and that these address operands must appear within the code and/or data segments. The code and data segments are scanned for values that lie within the range of the code segment. The resulting values are guaranteed to contain all of the indirect CTI targets.

The indirect CTIs are handled by appropriately translating the original target to the corresponding location in IR through a runtime translator. Each recognized procedure (through speculative disassembly) is initially considered a possible target of the translator, which is pruned further using alias analysis. The arguments for each possible target procedure are unioned to find the set of arguments to be passed to the translator; a stub inside the translator populates the arguments according to the actual target.

The method above is not sufficient for discovering indirect branch targets where addresses are calculated in binary. Hence, various procedure boundary determination techniques, like ending the boundary at beginning of next procedure, are also proposed [135] to limit possible targets.

The disassembler also implements several additional techniques [63] to recover procedure boundaries and inserts additional checks that are essential for the IR to be functional in case of inaccurate recovered boundaries.

3.3 Deconstruction of physical stack frames

In order to recover a source-level stack representation, we first recognize the local stack frame of a procedure and represent it as a local variable in the IR. As explained in Section 3.1, this local variable is coupled with the rest of the stack due to interprocedural accesses. We achieve this decoupling by recognizing interprocedural accesses and replacing them with symbolic accesses to the procedure arguments. Below, both these techniques are presented in detail.

3.3.1 Representing the local stack frame

We begin by finding an expression for the maximum size of the local stack frame in a procedure X . We analyze all the instructions which can modify the stack pointer, and find the maximum size, P , to which the stack can grow in a single invocation of procedure X among all its control-flow paths. P need not be a compile-time constant; a run-time expression for P suffices when variable-sized stack objects are allowed. An array `ORIG_FRAME` of size P is then allocated as a local variable at the entry point of procedure X in the IR.

The local variables for the frame pointer and stack pointer are initialized to the beginning of `ORIG_FRAME` at the entry point of procedure X . Thereafter, all the stack

pointer modifications — by constant or non-constant values — are represented as adjustments of these variables. Allocation of a single array representing the original local frame guarantees the correctness of stack arithmetic inside the procedure `X`.

In some procedures, it might not be possible to obtain a definite expression for the maximum size of the local stack frame. For example, scoped variable-sized local objects in source-code might result in a stack allocation with a non-constant amount, whose expression is not available at the beginning of the procedure. Consequently, a single array `ORIG_FRAME` of a definite size cannot be allocated. Neither can multiple local arrays, one per such stack increment, be allocated since IR optimizations and compiler backend can modify their relative layout thereby invalidating the stack arithmetic. In such procedures, we do not convert the physical stack to an abstract frame. A physical stack frame is maintained in the IR using inline assembly versions of all the stack modification instructions while the remaining instructions are converted to LLVM IR. The runtime checks mechanism presented in the next section is employed to distinguish the local and ancestor accesses.

Persistent stack modification: Returns from a procedure ordinarily restore the value of the stack pointer to the value before the call. However, in some cases, the stack pointer might point to a different location after returning from a procedure call. For example, the called procedure can cleanup the arguments passed through the stack. To represent this stack pointer modification, which persists beyond a procedure call, we introduce the following definition:

Balance Number: The balance number for a procedure is defined as the net

shift in the stack pointer from before its entry to after its exit. Four different cases can arise:

Case 1: *Balance Number* = 0

This is the common case; no modification required.

Case 2: *Balance Number* < 0

This case arises when a procedure cleans up a portion of the caller stack frame and is represented as an adjustment of the stack pointer by *Balance Number* amount in the caller procedure after the call. The amount need not be a constant.

Case 3: *Balance Number* > 0

This case implies that a procedure leaves its local frame on the stack and the corresponding frame outlives the activation of its procedure. Such procedures are represented by considering their allocation as part of the caller procedure allocation. The *Balance Number* amount is added to the size of `ORIG_FRAME` array in the caller procedure and the stack pointer is adjusted after the call by this amount.

Case 4: *Balance Number* Indeterminable

In such a case, we do not convert the physical frame into abstract frame and represent the stack as a default global variable in the IR, as shown in Fig 3.1(b). This is an extremely rare case and in fact, it did not appear in our experiments.

3.3.2 Representing procedure arguments

As per the source-level representation, we aim to represent all the stack-based interprocedural communication through an explicit argument framework. We discuss

why this is not feasible in all the cases and propose our novel methods based on run-time checks to handle such scenarios.

We use Value Set Analysis (VSA) [20] to aid our analysis. VSA determines an over-approximation of the set of memory addresses and integer values that each register and memory location can hold at each program point. Value Set (VS) of the address expression present in a memory access instruction provides a conservative but correct estimate of the possible memory locations accessed by the instruction. VSA accurately captures the stack pointer modifications and the assignments of stack pointer to other registers.

The stack location at the entry point of a procedure is initialized as the base (zero) in VSA and the local frame allocations are taken as negative offsets. Intuitively, memory accesses with positive offsets represent accesses into the parent frame and constitute the arguments to a procedure. A formal argument is defined for each constant offset into the parent frame and each such access is directly replaced by an access to the formal argument.

However, the above method for recognizing arguments is suitable only if VS of the address expression is a singleton set. If the VS has multiple entries, it is not possible to statically replace it with a single argument.

Fig 3.7 contains an x86 assembly fragment which will be used to illustrate the handling of interprocedural accesses. Fig 3.8 shows the output IR that results from Fig 3.7.

We introduce the following definitions to ease the understanding:

```

1. function foo:
2.  sub 100, esp      // Subtract 100 from sp
3.  call bar         // call bar

4. function bar:
5.  sub 10, esp      // Subtract 10 from sp
6.  lea 4(esp),edi  // Move address esp+4 to edi
7.  mov 2, ebx       // Move value 2 to ebx
8.  mov 15, ecx      // Move value 15 to ecx
9.  if (esi < 5) jmp B2 //Conditional Branch

10. B1: mov 4,ebx    // Move value 4 to ebx
11.  mov 16,ecx     //Move value 16 to ecx

12. B2: store 10, ebx[edi] // Store 10 to indirect offset (edi + ebx)
13.  store 10, ecx[esp] // Store 10 to indirect offset (esp + ecx)
14.  store 10, edx[edi] // Store 10 to indirect offset (edi + edx)

```

Figure 3.7: A small pseudo-assembly code. The second operand in the instruction is the destination.

`CURRENT_BASE`: Stack pointer at the entry point of a procedure.

`addrm`: The address expression of a memory access instruction `m`

`VS(addrm)`: Value Set of `addrm`

`(x,y)`: Lower and upper bounds, respectively, of the possible offsets relative to `CURRENT_BASE` in `VS(addrm)`

`LOCAL_SIZE`: Size of local frame variable `ORIG_FRAME`

`SIZEi`: Size of `ORIG_FRAMEi` of the ‘`i`th’ ancestor in the call graph, with the caller being represented as the first ancestor. `SIZE0` is defined as value 0.

Three different cases for memory reference categorization of a memory access instruction `m` arise:

Case 1: $(x,y) \subset (-LOCAL_SIZE,0)$

This condition implies that the current memory access instruction strictly refers to a local stack location. In Fig 3.7, Line 12 corresponds to this case. Instruction at Line 6 moves address `esp+4` to register `edi`. Since the size of the current frame in `bar` (`LOCAL_SIZE`) is 10 and the local allocations are taken as negative offsets, this translates to VS of `edi` as `{CURRENT_BASE-6}`. The VS of `ebx` at Line 12 is `{2,4}`; therefore the `VS(addrm)` is `{CURRENT_BASE-2, CURRENT_BASE-4}`, which translates as a subset of `(-LOCAL_SIZE, 0)`. In this case, we replace the indirect access by an access to the local frame as shown Fig 3.8 (Line 12).

Case 2: $\exists N : (\mathbf{x}, \mathbf{y}) \subset (\sum_{i||i \in (0, N)} \mathbf{SIZE}_i, \sum_{i||i \in (0, N+1)} \mathbf{SIZE}_i)$

This case implies that the current instruction exclusively accesses the local frame of N^{th} ancestor. In such cases, we make the local frame variable of the N^{th} ancestor procedure, `ORIG_FRAMEN`, an extra incoming argument to the current procedure as well as to all the procedures on the call-graph paths from the ancestor to the current procedure. The indirect stack access is replaced by an explicit argument access.

Line 13 in Fig 3.7 represents this case. Here, VS of `ecx` is `{15,16}` which translates to the stack-offset range `(5,6)` which is subset of `(0, SIZE1)`. Line 13 in Fig 3.8 shows the adjusted offset into the formal argument `inArg`.

Case 3:

$\exists N : \{ \{ (\mathbf{x}, \mathbf{y}) \cap (\sum_{i||i \in (0, N)} \mathbf{SIZE}_i, \sum_{i||i \in (0, N+1)} \mathbf{SIZE}_i) \neq \emptyset \} \wedge \{ (\mathbf{x}, \mathbf{y}) \not\subset (\sum_{i||i \in (0, N)} \mathbf{SIZE}_i, \sum_{i||i \in (0, N+1)} \mathbf{SIZE}_i) \} \}$

This case arises when VSA cannot bound the memory access exclusively to

```

1.function foo:
2.  ORIG_FRAME_FOO=alloca i32, 100 // Local frame allocation
3.  call bar(ORIG_FRAME_FOO)      // call bar

4.function bar(i32* inArg)
5.  ORIG_FRAME_BAR=alloca i32, 10  // Local frame allocation
6.  edi = ORIG_FRAME_BAR+4
7.  ebx = 2                        // Move value 2 to ebx
8.  ecx = 15                       // Move value 15 to ecx
9.  if (esi < 5) jmp B2

10. B1: ebx = 4                    // Move value 4 to ebx
11.  ecx = 16                      // Move value 16 to ecx

12. B2: store 10, ebx[edi]         // Store 10 to local frame
13.  store 10, (ecx-SIZE_BAR)[inArg] // Ancestor Store
14.  if ((edx+edi - ORIG_FRAME_BAR) < SIZE_BAR) //Run Time Check
15.    store 10, edx[edi]          //Local Store
16.  else
17.    store 10, (edx+edi - SIZE_BAR)[inArg] //Ancestor Store

```

Figure 3.8: *IR of the pseudo-assembly code. $SIZE_BAR$ is size of $ORIG_FRAME_BAR$, register names are pure IR symbols.*

the local frame of one ancestor or to the local frame of the current procedure. It also includes cases where VS of the target location is *TOP* (i.e., unknown).

We propose a run-time-check-based solution to represent such accesses in the IR. We define all the possible ancestor stack frames in the call graph as arguments to this procedure. Further, at the indirect stack access, a run-time check is inserted in the IR to dynamically translate the access to the local frame or to one of the ancestor stack frames.

Line 14 in Fig 3.7 represents this case. Suppose edx is data-dependent and hence its VS is *TOP*. Line 14 in Fig 3.8 shows the run-time check inserted based on this value. Depending on this check, we either access the local frame (Line 15) or the incoming argument (Line 17).

We have neglected the return address buffer in our calculations for ease of un-

derstanding. It is easily considered in our model by adding the return buffer size to each ancestor’s local frame size. In the case of dynamically linked libraries (DLLs), the procedure body is not available; hence the above method for handling the arguments cannot be applied. In order to make sure that the external procedures access arguments as before, the LLVM code generator is minimally modified to allocate the abstract frame, `ORIG_FRAME`, at the bottom of the stack in each procedure in the rewritten binary. Since external procedures are not aware of the call hierarchy inside a program, their interprocedural references are usually limited to only the parent frame. When the prototypes of these external procedures are available (such as for standard library calls), this stack maintenance restriction is avoided altogether by employing the solution presented for any other procedure.

3.4 Translating memory locations to symbols

Section 3.3 presented methods for deconstructing the physical stack frame into individual abstract frames, one per procedure. Even though this representation allows unrestricted modification of the stack frame, accesses to local variables appear as explicit memory references to locations within this array, which are not amenable to standard dataflow analysis. In this section, we propose our methods for translating these memory operations to symbol operations in the IR.

1. store eax, ebx[esi]	1. store eax, ebx[esi]
2. load 8[esp], edx	load 8[esp],sym	1. store eax, 8[esp]
.....	2. mov sym, edx	2. load 8[esp], edx
3. store ecx, 8[esp]	3. load ebx[esi],edx
....	3. mov ecx, sym
4. load 8[esp], edi	4. store eax,ebx[esp]
5. load ebx[esi], edx	4. mov sym, edi	5. load 8[esp], ecx
	store sym, 8[esp]	6. load ebx[esi],edx
	5. load ebx[esi], edx
a)	b)	c)

Figure 3.9: *Symbol promotion. Second operand in the instruction is the destination of the instruction.*

3.4.1 Motivation for partitions

As discussed in Section 3.1, maintaining data-flow consistency of the underlying memory locations across the whole program is imperative while promoting memory accesses to symbolic accesses. Fig 3.9(a) shows a small example with three direct accesses to location (`esp+8`) at Lines 2,3,4; the remaining two are unbounded indirect accesses. The simplest method for maintaining the data-flow consistency across the program is to load the data from the memory location into the symbol just after each aliasing definition, store the symbol back to the memory location just before each aliasing use and promote each candidate stack access to a symbolic access, as shown in Fig 3.9(b). The load inserted just after the aliasing definition is referred to as a *Promoting Load* and the store just before the aliasing use is referred to as a *Promoting Store* (shown as bold in Fig 3.9(b)). Although this method ensures correct data flow propagation, it results in a large number of promoting loads and stores which might overshadow the benefit of symbol promotion.

Fig 3.9(c) illustrates this unprofitable case. In this example, suppose VS of `ebx` is TOP. Consequently, the instructions at Line 3, 4 and 6 are aliasing indirect accesses to the stack location (`esp+8`). In order to promote the direct memory accesses at instructions 1, 2 and 5, we need to insert Promoting Stores just before instruction 3 and instruction 6 and a Promoting Load just after instruction 4. Hence, promoting three direct memory operations entails the insertion of three extra memory operations, nullifying the benefit.

We propose a novel partition-based symbol promotion algorithm where we divide the program into a set of non-overlapping promotional lifetimes for each memory location. It serves as a fine-grained framework where the symbol promotion decision can be made independently for each lifetime (a partition) instead of the entire program at once. Not doing symbol promotion in a partition does not affect the correctness of the data-flow in the program. The symbol promotion can be selectively performed in only those partitions where it is provably beneficial. Fig 3.9(c) shows an intuitive division of the current example into two safe partitions.

3.4.2 Reaching definition framework

We define a new reaching definition analysis on *memory locations* for computing the partitions. This is different from the standard reaching definitions on *symbols* well-known in compiler theory. For each memory location `loc`, this analysis computes the set of instructions defining the memory location `loc` that reach each program point. The set of definitions includes stores to the memory location `loc` using direct

Statement s	$gen[s]$	$kill[s]$
$d: store\ x, mem[reg]$	$if([sp+addr] \in VS(mem+reg))$ d $else\ \{ \}$	$if([sp+addr] \in VS(mem+reg))$ $defs(addr) - d$ $else\ \{ \}$
$d: store\ y, addr[sp]$	d	$defs(addr) - d$
$d: z = load\ mem[reg]$	$\{ \}$	$\{ \}$
$d: z = load\ addr[sp]$	$\{ \}$	$\{ \}$

Memory location $loc : [sp + addr]$

mem : Non-constant access

$addr$: Constant

$defs(addr)$: Set of instructions defining the memory location $[sp+addr]$

$in[n]$: Set of definitions that reach the beginning of node n

$out[n]$: Set of definitions that reach the end of node n

$pred[n]$: Predecessor nodes of node n

$in[n] = \cup_{i \in pred[n]} \{out[p]\}$

$out[n] = gen[n] \cup (in[n] - kill[n])$

Figure 3.10: *The reaching definition description. Definitions are propagated across the control flow of program.*

addressing mode as well as possibly aliasing stores.

Fig 3.10 formulates the reaching definition in terms of VS of the memory accesses. These reaching definitions are propagated across the control flow of the program, similar to the standard compiler dataflow propagation, allowing the partitions to be formed across basic blocks. The interprocedural version of VSA implicitly takes into consideration a local pointer passed to a procedure through an argument.

3.4.3 Symbol promotion algorithm

The candidates for symbol promotion in a procedure P , represented by a set $LDCS$, are computed as follows:

M: Set of memory accesses in P

DM: Statically determinable memory accesses, $\bigcup_{d \in M} \{d \mid \|VS(addr_d)\| = 1\}$

LOCS: Statically determined stack locations in P, $\bigcup_{d \in DM} \{m \mid m \in VS(d)\}$

Mathematically, for a stack location `loc`, a single partition constitutes three sets of memory accesses: *DirectAcc*, *BeginSet* and *EndSet*. *DirectAcc* contains statically determinable accesses to the location `loc` and constitutes the potential candidates for symbol promotion. *BeginSet* constitutes the indirect stores that may-alias with `loc` and have a control flow path to at least one element of the set *DirectAcc*. *EndSet* consists of all the aliasing accesses such that there is a control flow path from some element of *BeginSet* to these accesses. Intuitively, program points just after the elements in *BeginSet* represent the locations for inserting Promoting Loads. Similarly, program points just before the elements of *EndSet* are the locations for inserting Promoting Stores.

Algorithm 1 provides a formal description of the method for computing partitions for a memory location `loc`. We begin with an empty partition. We analyze a store instruction, say `ds`. If `ds` is a direct addressing mode instruction then it is added to the *DirectAcc* set; otherwise it is added to *BeginSet* (Line 9-12). Load instructions using direct addressing where `ds` is one of the reaching definitions are added to the *DirectAcc* set of the partition (Line 16-18). The remaining reaching definitions at these load instructions are added to the analysis list (Line 19-20). If `ds` uses a direct addressing mode, indirect load and store instructions with `ds` as one of the reaching definitions are added to the *EndSet* (Line 24-26). For indirect

```

1 L: Set of loads in P; S: Set of stores in P
2 DL:  $\bigcup_{l \in L} \{l \mid \{loc\} = VS(addr_l)\}$  //Direct Loads
3 IL:  $\bigcup_{l \in L} \{l \mid \{loc\} \subset VS(addr_l)\}$  //Indirect Aliasing Loads
4 DS:  $\bigcup_{s \in S} \{s \mid \{loc\} = VS(addr_s)\}$  //Direct Stores
5 IS:  $\bigcup_{s \in S} \{s \mid \{loc\} \subset VS(addr_s)\}$  //Indirect Aliasing Stores
6 Processed: Set of elements processed
7 while DS !=  $\emptyset$  || IS !=  $\emptyset$  do
8   define new Partition P, define new list ActiveList
9   if DS !=  $\emptyset$  then
10    | s = DS.begin; add s to P.DirectAcc
11   else
12    | s = IS.begin; add s to P.BeginSet
13   add s to ActiveList
14   while ActiveList.size != 0 do
15    | s = ActiveList.top; Add s to Processed
16    | for dl  $\in$  DL do
17      | if s  $\in$  in[dl] then
18        | add dl to P.DirectAcc
19        | for s'  $\in$  in[dl] do
20          | add s' to ActiveList if s'  $\notin$  Processed
21        | remove dl from DL
22    | if s  $\in$  IS then
23      | continue /* No need to store symbol back */
24    | for il  $\in$  {IL, IS} do
25      | if s  $\in$  in[il] then
26        | add il to P.EndSet
27      | for s'  $\in$  in[il] do
28        | add s' to ActiveList if s'  $\notin$  Processed
29      | remove il from IL if il  $\in$  IL

```

Algorithm 1: *Algorithm for computing partitions for a location loc in a procedure P*

stores, the symbol need not be stored back to the memory (Line 22-23). As with the direct loads, the rest of the reaching definitions are added to the analysis list (Line 27-29). This analysis is applied repeatedly until the analysis list is empty. At that point, we have one independent partition. We repeatedly obtain new partitions until there are no more direct stores or indirect stores to analyze.

We implement a simple benefit-cost model to determine whether the symbol promotion should be carried out for a particular partition. In a partition, the size of DirectAcc set is the number of memory accesses replaced by symbol accesses. We define $Freq_i$ as the statically determined execution frequency at program point i . Hence, the benefit of symbol promotion in terms of eliminated memory references:

$$Benefit = \sum_{i \in DirectAcc} \{(Freq_i)\}$$

One promoting load/store is needed for each element of BeginSet and Endset, consequently, the cost:

$$Cost = \sum_{i \in BeginSet} \{(Freq_i)\} + \sum_{i \in EndSet} \{(Freq_i)\}$$

We calculate the net benefit of each partition as $Benefit - Cost$. Symbol promotion is carried out in a partition only if the net benefit is positive.

3.5 Results

Table 3.1 lists all the benchmarks which have been successfully evaluated with the SecondWrite prototype. It includes SPEC2006 benchmark suite, benchmarks from other suites and a real world program, Apache server. Benchmarks on Linux are compiled with gcc v4.4.1 (O0 (No optimization) and O3 (Full optimization) flags) without any symbolic or debug information. Windows benchmarks are compiled with Microsoft Visual Studio compiler (O0 (No optimization) and O2 (Maximum optimization) flags). Only the C and C++ programs are included for Windows since Visual Studio does not compile Fortran. The benchmarks are compiled for x86-32 ISA and results are obtained for SPEC2006 *ref* datasets on a 2.4GHz 8-

Application	Source	Lang	LOC	Platform	Compiler	# of Func	Stack locations promoted	Stack access promoted
bwaves	Spec2006	F	918	Linux	gcc-00	8	78.7	86.9
					gcc-03	8	86.9	93.5
lbm	Spec2006	C	1155	Linux	gcc-00	20	77.3	85.6
					gcc-03	15	72.4	71.7
equake	OMP2001	C	1607	Linux	gcc-00	23	62.1	87.5
					gcc-03	10	34.5	47.5
art	OMP2001	C	1914	Linux	gcc-00	26	87.2	95.9
					gcc-03	17	93.7	76.2
wupwise	OMP2001	F	2468	Linux	gcc-00	28	75.2	97.2
					gcc-03	28	66.3	82.2
mcf	Spec2006	C	2685	Linux	gcc-00	23	99	98.5
					gcc-03	21	99.1	93.9
namd	Spec2006	C++	3188	Linux	gcc-00	138	40.7	42.8
					gcc-03	89	33.9	48.3
leslie3d	Spec2006	F	3807	Linux	gcc-00	20	79.5	99.6
					gcc-03	20	84.5	94.5
libquant	Spec2006	C	4357	Linux	gcc-00	94	90.5	90.5
					gcc-03	66	80.4	75.8
astar	Spec2006	C++	5842	Linux	gcc-00	129	72.1	86.2
					gcc-03	60	77.1	92.3
bzip2	Spec2006	C	8293	Linux	gcc-00	106	77	82.5
					gcc-03	51	68.9	88.5
milc	Spec2006	C	9784	Linux	gcc-00	192	64.7	81.9
					gcc-03	164	73.3	76.2
sjeng	Spec2006	C	13847	Linux	gcc-00	130	58.2	83.4
					gcc-03	103	36.2	38.7
sphinx	Spec2006	C	13683	Linux	gcc-00	268	72.8	78.5
					gcc-03	199	65.1	67.5
zeusmp	Spec2006	F	19068	Linux	gcc-00	55	97.2	97.2
					gcc-03	55	57.3	82.2
omnetpp	Spec2006	C++	20393	Linux	gcc-00	2691	64.61	65.8
					gcc-03	2052	84.6	60.1
hmmr	Spec2006	C	35992	Linux	gcc-00	331	84.3	84.3
					gcc-03	295	79.6	77.5
solpex	Spec2006	C++	28592	Linux	gcc-00	1548	85.7	88
					gcc-03	1249	72.3	75.3
h264	Spec2006	C	51578	Linux	gcc-00	885	65.9	79
					gcc-03	775	61.1	70
cactus	Spec2006	C/F	60452	Linux	gcc-00	752	70.9	75.5
					gcc-03	625	70.9	69.8
gromacs	Spec2006	C/F	65182	Linux	gcc-00	1126	76.4	75
					gcc-03	598	87.6	74.5
deall	Spec2006	C++	96382	Linux	gcc-00	1482	72.3	74.4
					gcc-03	1283	65.1	68.2
calculix	Spec2006	C/F	105683	Linux	gcc-00	887	61.3	66.5
					gcc-03	766	56.6	67.9
tonto	Spec2006	F	108330	Linux	gcc-00	1971	47.6	54.54
					gcc-03	1861	53.5	57.8
povray	Spec2006	C++	108339	Linux	gcc-00	2063	71.4	70.8
					gcc-03	1601	60.1	66.2
gobmk	Spec2006	C	157883	Linux	gcc-00	2594	78.6	81.4
					gcc-03	2391	62.16	65.7
perlbench	Spec2006	C	126367	Linux	gcc-00	1686	76.4	74.1
					gcc-03	1459	33.1	30.7
gcc	Spec2006	C	236269	Linux	gcc-00	5206	66.1	72.2
					gcc-03	4897	63.1	66.2
lbm	Spec2006	C	1155	Windows	VS-00	21	67.1	73.1
					VS-02	19	95.5	66.1
equake	OMP2001	C	1607	Windows	VS-00	26	53.5	80.5
					VS-02	14	31.4	32.2
art	OMP2001	C	1914	Windows	VS-00	26	78.6	85.5
					VS-02	25	74.6	77.3
mcf	Spec2006	C	2685	Windows	VS-00	24	93.9	90.9
					VS-02	22	81.9	79.3
namd	Spec2006	C++	3188	Windows	VS-00	128	37.5	56.1
					VS-02	136	35.4	36.2
astar	Spec2006	C++	4377	Windows	VS-00	133	80.1	81.1
					VS-02	121	70.1	76.9
bzip2	Spec2006	C	5896	Windows	VS-00	85	69.1	57.8
					VS-02	67	70.9	65.4
milc	Spec2006	C	9784	Windows	VS-00	188	76.39	66.1
					VS-02	144	63.31	75.91
sjeng	Spec2006	C	13847	Windows	VS-00	128	37.5	56.1
					VS-02	107	59.5	48.3
sphinx	Spec2006	C	13683	Windows	VS-00	264	53.5	80.5
					VS-02	241	66.1	67.1
omnetpp	Spec2006	C++	20393	Windows	VS-00	2281	63.1	67.2
					VS-02	2061	59.1	60.3
hmmr	Spec2006	C	35992	Windows	VS-00	267	76.9	79.94
					VS-02	263	69.9	80.82
h264	Spec2006	C	51578	Windows	VS-00	546	58.9	67.3
					VS-02	476	54.71	51.02
perlbench	Spec2006	C	126367	Windows	VS-00	1826	54.4	58.17
					VS-02	1696	55.2	60.1
gobmk	Spec2006	C	157883	Windows	VS-00	2589	63.3	43.5
					VS-02	2382	57	29.1
gcc	Spec2006	C	236269	Windows	VS-00	5109	62.1	57.1
					VS-02	4872	64.5	67.1
apache server	Real World Program	C	232931	Linux	gcc-00	2459	63.2	68.1
					gcc-03	2046	75.1	79.2
TOTALS			2 Million			AVG	67.84867	71.733333

Table 3.1: *Benchmarks Table*

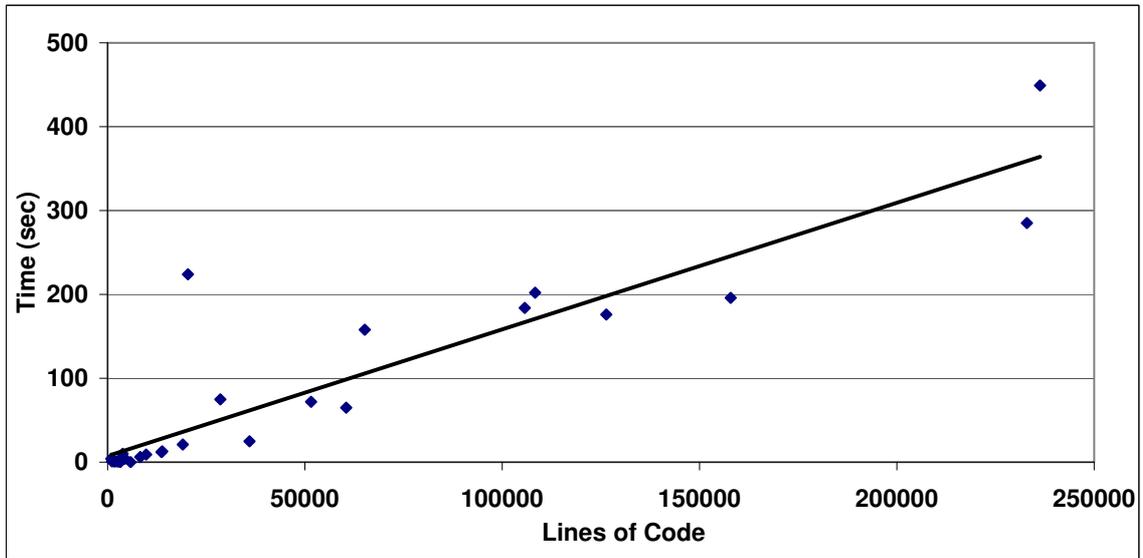


Figure 3.11: *Variation of analysis time with lines of code. Outlier program dealII has been omitted for the ease of presentation.*

core Intel Nehalem machine running Ubuntu. The performance analysis of Apache server is carried out using *ab* tool [12]. Analyzing executables compiled by a new compiler causes several engineering challenges with our evolving prototype such as presence of yet unsupported x86 features like SSE and other advanced instructions. However, successful experimentation with distinct compilers such as gcc and Visual Studio demonstrates the lack of any fundamental problem in this regard. In future work, we aim to expand our support base by evaluating executables compiled by other compilers such as Intel compiler and LLVM. Unless mentioned explicitly, the benchmarks in figures are the ones compiled by gcc.

Fig 3.11 plots the variation in the time taken by SecondWrite, with increasing lines of code, to recover an intermediate representation from an executable. This constitutes the time spent in disassembling the executable and other analyses including abstract stack recovery and symbol promotion. Fig 3.11 highlights the nearly

linear scalability of our framework. The analysis time for large programs such as *gcc*, containing 250,000 lines of code, is around eight minutes. A particular SPEC benchmark *dealIII* takes around 35 minutes, forming an outlier to the linear model. It employs templates excessively which causes the compiler to create multiple versions of the same procedure for different template parameters. This extensively slows down several interprocedural analyses resulting in a huge overall analysis time.

3.5.1 Static characteristics

Our symbol promotion techniques promote the stack memory locations to symbols and direct stack memory accesses to symbol accesses in the IR. On average, 67% of stack locations are promoted to symbols resulting in promotion of 72% of direct stack accesses for the programs listed in Table 3.1. For the remaining memory operations, the net benefit for promotion didn't meet the corresponding threshold. Theoretically, our framework can achieve *100% symbol promotion* if the promotion threshold is ignored, but this leads to high overhead in the rewritten binaries due to *Promoting Loads* and *Promoting Stores*. The development of more advanced alias analysis would improve results of our symbol promotion without adversely affecting the performance.

Fig 3.12 relates the above promoted symbolic references to the original source-level artifacts. We enumerated the symbolic references in the input program using debug information (employed only for counting the references) and compared how many of these symbolic references are restored in the IR. It shows that our techniques

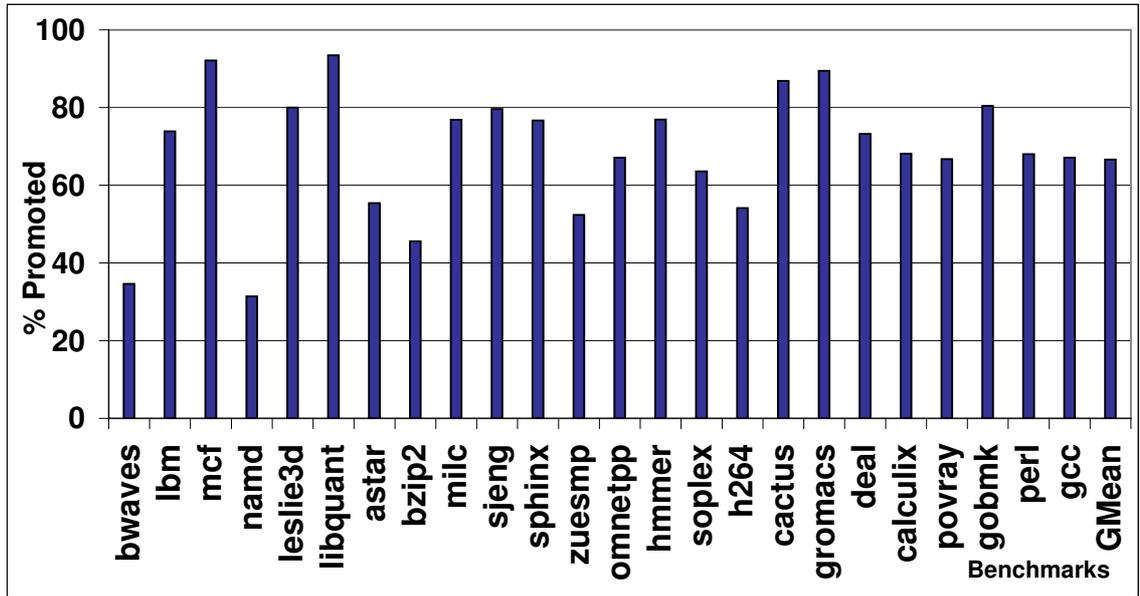


Figure 3.12: *Percentage of original symbolic accesses recovered in IR.*

Program	Version	# Proc with Physical stack	# Proc with runtime Checks
gcc	gcc-O0,VS-O0	117	0
gcc	gcc-O3, VS-Ox	117	10
tonto	gcc-O0, gccO3	20	0

Table 3.2: *Corner cases of our analysis.*

are able to restore 66% of the original symbolic references.

Fig 3.13 presents an insightful result regarding our partition algorithm (Alg 1). Our partitioning algorithm creates fine-grained promotional lifetimes for each memory location. On average, around 76% of the memory locations have one partition, 18% have two to five, and 6% have five or more partitions. This is not unexpected since large procedures are relatively rare.

Table 3.2 lists the programs which hit corner cases during the deconstruction of physical stack. The analysis of the original source-code revealed that a physical stack frame was required for procedures that call *alloca()*. Runtime checks

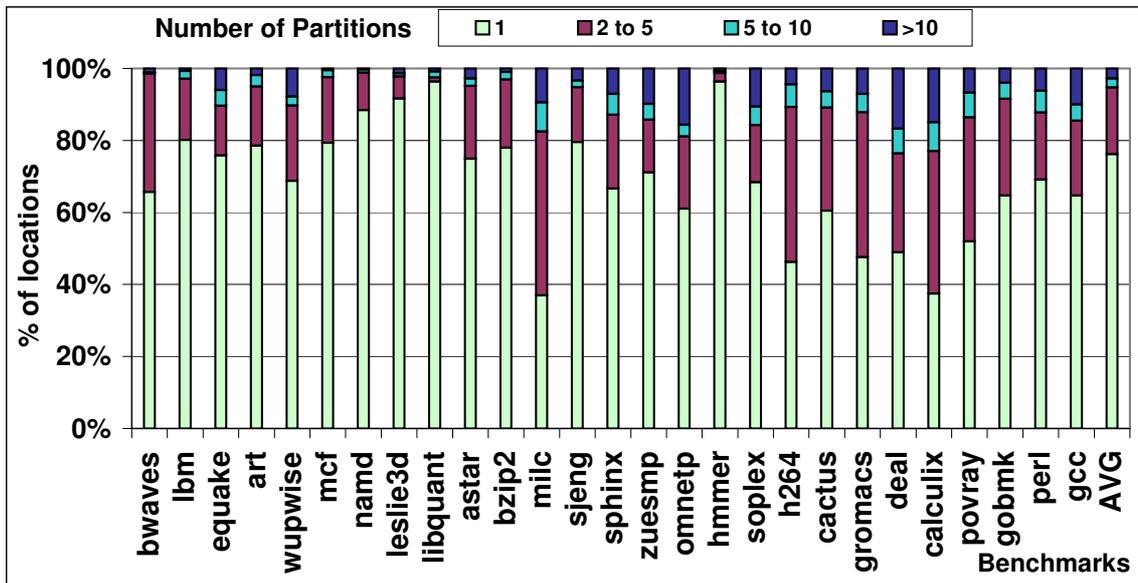


Figure 3.13: *Partition algorithm visualization*

are inserted in some procedures which accept a variable number of arguments using the *va_arg* mechanism. Most of the procedures using *va_arg* do not require runtime checks. This result establishes our earlier hypothesis that scenarios requiring run-time checks are extremely rare and consequently, have negligible overhead. Nonetheless, not handling these scenarios prohibits obtaining a functional IR and hence, are imperative for any translation system.

3.5.2 Un-optimized input binaries

Fig 3.14 shows the normalized run-time of each rewritten binary compared to an input binary produced using gcc with no optimization (-O0 flag). Fig 3.15 shows the corresponding run-time for binaries produced using Visual Studio compiler with no optimization (-O0 flag). We obtain an average improvement of 40% in execution time for binaries produced by gcc and 30% for binaries produced by Visual Studio, with

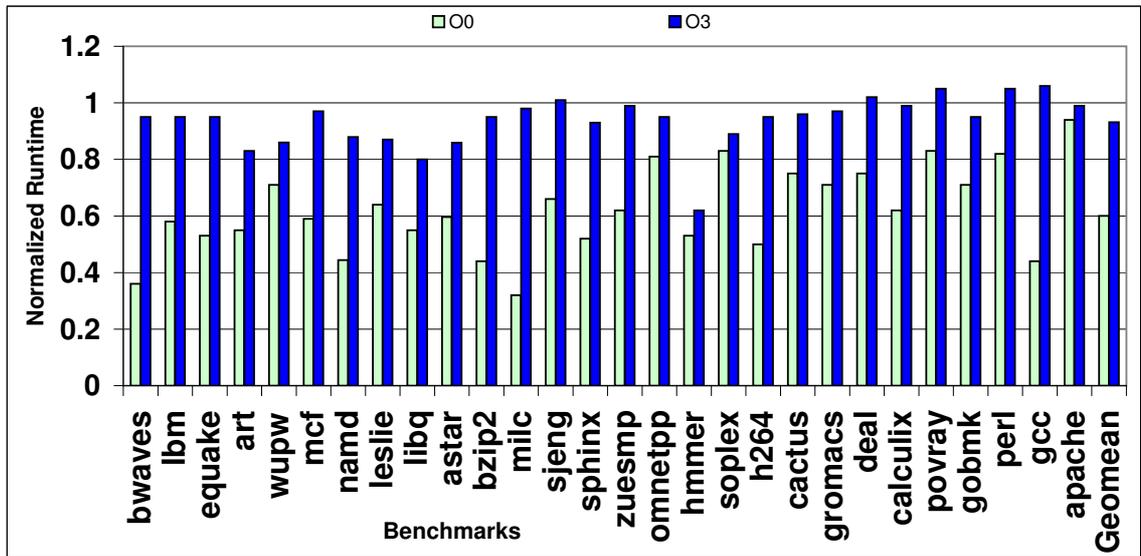


Figure 3.14: Normalized runtime of rewritten binary as compared to its corresponding input version (=1.0) compiled by gcc.

an improvement of over 65% in some cases (*bwaves*). In fact, as shown in Fig 3.16, our tool brings down the normalized runtime of unoptimized input binaries from 2.2 to close to the runtime (1.25) of gcc-optimized binaries.

3.5.3 Optimized input binaries

Fig 3.14 shows the normalized execution time of each rewritten binary compared to an input binary produced using gcc with the highest-available level of optimization (-O3 flag). In this case, we obtain an average improvement of 6.5% in execution time. It is interesting that we were able to obtain this improvement over already optimized binaries without any custom optimization of our own. One of our rewritten binaries (*hmmr*) had a 38% speedup vs the input binary. Although gcc -O3 is known to produce good code, it missed the creation of few predicated instructions whereas LLVM did this optimization, explaining the speedup. Fig 3.15 shows the

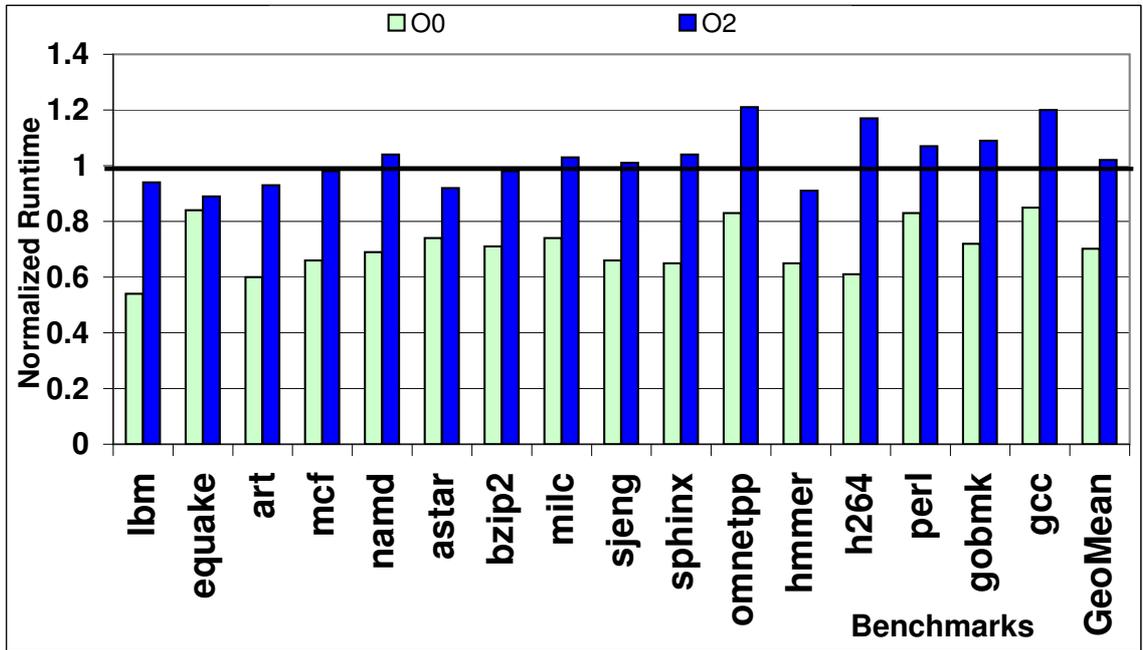


Figure 3.15: *Normalized runtime of rewritten binary as compared to its corresponding input version (=1.0) compiled by Visual Studio.*

corresponding run-time for binaries produced using Visual Studio compiler with full optimization flag (-O2). As evident, our framework was able to retain the performance of these binaries, with a small overhead of 2.7% on average.

3.5.4 Impact of symbol promotion

Next, we substantiate the impact of symbol promotion on the run-time of rewritten binaries. Fig 3.17 and Fig 3.18 show the normalized improvement in execution time obtained by applying only LLVM optimizations and by applying our symbol promotion techniques. It shows that symbol promotion is responsible for improving the average performance of rewritten binary from 30% to 40% in the case of unoptimized binaries (produced by gcc) and from 1% to 6.5% in the case of optimized binaries (produced by gcc). Since our cost metric is based on static profiling, we

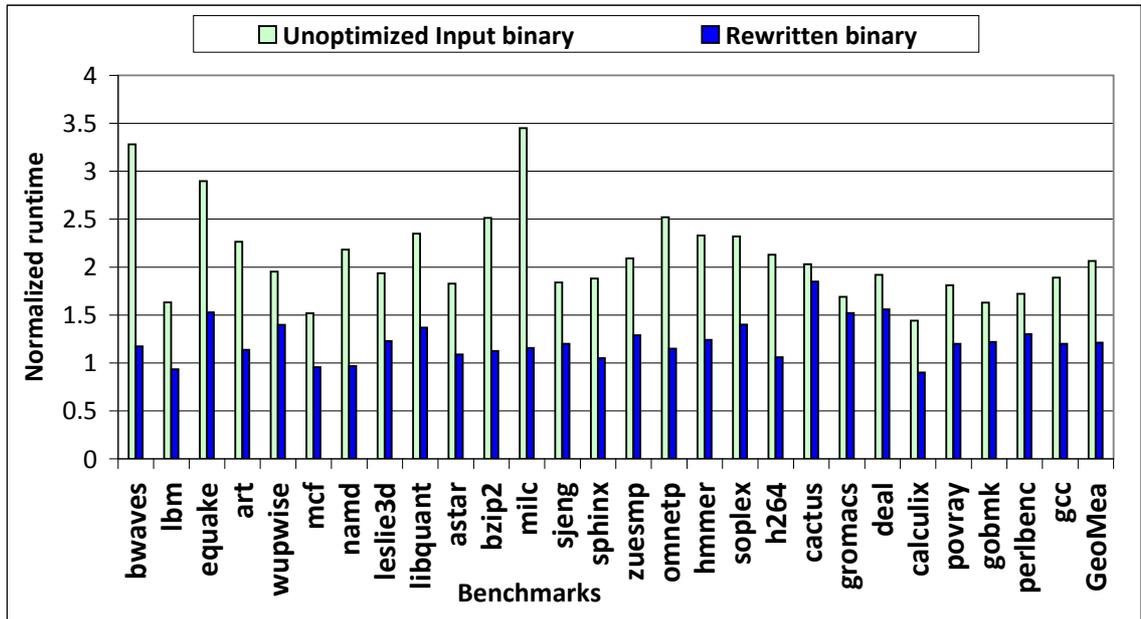


Figure 3.16: Normalized runtime of rewritten binary as compared to optimized version (=1.0) compiled by gcc.

observed a small slowdown with symbol promotion in *bzip2 O3*.

It is important to note that these results only measure the impact of symbol promotion. The impact of our method to convert physical frames to abstract frames is not measured above. However, we can infer that number since without obtaining abstract frames, none of the existing LLVM passes would run at all, leading to zero run-time improvement.

3.5.5 Symbolic Execution

KLEE is efficiently designed to obtain a high code coverage on source programs. We run KLEE in our framework on a set of 50 alphabetically-chosen Coreutils executables and achieve a code coverage of 73% on average compared to 76% obtained by KLEE on source programs when running KLEE with the same options and for the

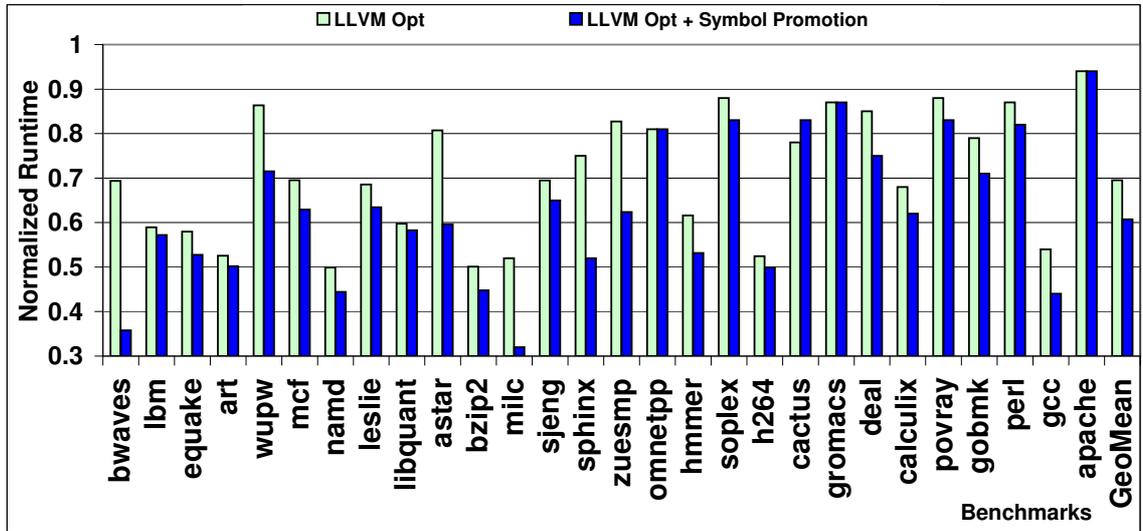


Figure 3.17: *Impact of symbol promotion on runtime of rewritten binary v/s unoptimized input binary (=1.0).*

same amount of time (30 minutes/benchmark) in both cases.

Recall from Section 3.1.1.1 that symbol promotion enables our framework to efficiently reason about symbolic memory accesses. However, most of the Coreutils programs do not contain symbolic array accesses, consequently, these programs are not likely to benefit from our analyses. Instead, a set of programs [42] with known symbolic accesses were chosen to demonstrate the effectiveness of our analysis. Each application was run with KLEE without symbol promotion for five minutes. Then, the applications were run with symbol promotion with the exact same workload. As evident from Table 3.3, our analysis is highly effective in reducing the time spent by STP solvers in query processing.

KLEE has been shown to detect various bugs in a particular version of Coreutils (6.10). Our framework enables the detection of these bugs from their corresponding executables. Further, the presence of a rewriting path in our framework

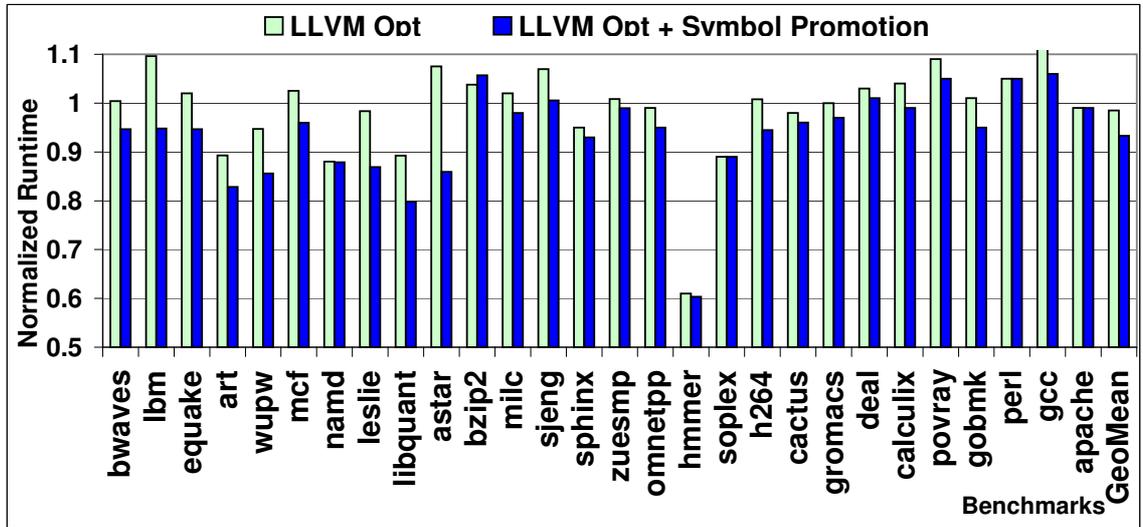


Figure 3.18: Impact of symbol promotion on runtime of rewritten binary v/s optimized input binary (=1.0).

Binary	No Promotion		With Promotion	
	Time(s)	STP Time(s)	Time(s)	STP Time(s)
htget	300	186	37	27
cut	300	252	111	76
split	300	225	157	88

Table 3.3: Improvement in constraints processing with symbol promotion.

enables us to remedy the above detected bugs directly from executables. We analyzed the dump for one of the Coreutil executable (*mkdir*), fixed the corresponding behavior in IR and obtained a rewritten bug-free executable.

3.5.6 Automatic Parallelization

Kotha et al [93] presented a method for automatic parallelization for binaries. Here, we substantiate the impact of symbol promotion on their methods for a subset of *PolyBench* and *Stream* suite. Fig 3.19 shows that symbol promotion increases the speedup of eleven benchmarks by 2.25x for four threads.

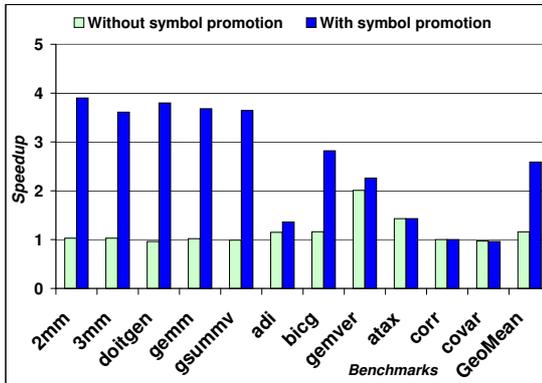


Figure 3.19: *Automatic parallelization*

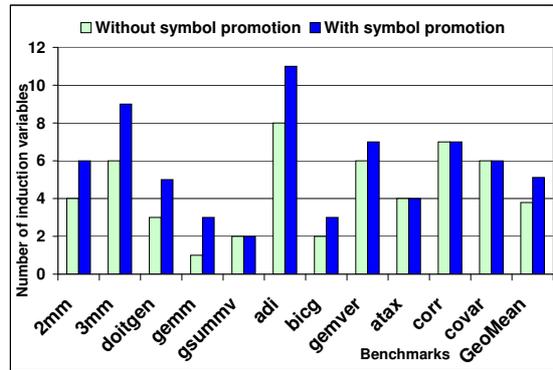


Figure 3.20: *Number of induction variables recognized*

We further investigate why symbol promotion helped automatic parallelization significantly. In order to parallelize loops using an affine automatic parallelizer, it is essential to recognize induction variables. We observe that for x86 binaries, many induction variables are often present on the stack instead of registers; the compiler’s induction variable recognizer based on symbols fails to recognize them. Further, for affine loops of nesting depth greater than two, induction variables of outer loops are generally placed on the stack. This results in parallelization of only inner loops even though outer-loop parallelization is legal. Parallelizing inner loops implies that there is a significant overhead due to synchronization and hence the speedup is low. On the other hand, symbol promotion promotes the stack allocated-induction variables corresponding to outer loops also to symbols; consequently, these induction variables get recognized and it allows the parallelizer to do parallelization on more beneficial outer loops. Detailed statistics of the number of outer loops for which induction variables are recognized with and without symbol promotion are presented in Fig 3.20.

Chapter 4: **Symbolic Analysis for executables**

4.1 Introduction

Analyzing and optimizing programs from their executables has received a lot of attention recently in the research community. The additional advantages offered by binary-level tools over traditional source-code level frameworks is the prime reason for this great interest. Binary-level tools can be employed to analyze executables produced by any compiler, can be applied in the absence of source code (legacy binaries) and can be used by an end-user for custom security analysis and platform-specific optimizations.

A typical source-code level framework employs various static analyses for analyzing and optimizing programs. Symbolic analysis [76, 31, 77, 32] is an important source-code level static analysis technique which represents the values of program variables through symbolic expressions. A symbolic analysis tool interprets programs in an abstract symbolic domain, maps each operation in its concrete domain to a corresponding operation in the symbolic domain and discovers the program properties using abstract characteristics. Symbolic analysis methods have been used regularly in traditional optimizing compilers. For example, optimizations like com-

mon subexpression elimination and global value numbering determine an equivalence of two symbolic expressions and remove the redundant computations [7]. Symbolic analysis has also emerged as an advanced technique to support the detection of parallelism in programs. Various parallelizing compilers [76, 31, 77] have employed powerful symbolic analysis frameworks to resolve data-dependency queries and to enable critical optimizations like array privatization for effective parallelization of programs.

However, the symbolic analysis frameworks employed in existing compilers operate only on the program variables. These frameworks handle memory accesses and memory locations in a very conservative manner. This is not suitable for executables since executables do not contain explicit program variables and store many of their variables in memory locations. This problem is exacerbated in the x86 ISA since its register set is very small, and hence most locations are memory-allocated. Hence, existing symbolic analysis methods have limited effectiveness when applied to executables.

There have been a very few methods for employing symbolic analysis for executables. Debray et al [58] suggested an alias analyses technique based on an underlying symbolic analysis framework. Amme et al [8] proposed a similar symbolic analysis framework for data dependence analysis of assembly code. However, both these techniques suffer from a major limitation. They restrict their analysis to registers only; they do not track symbolic values corresponding to the contents of the memory locations. Consequently, they lose a great deal of precision at each memory access. This severely limits their ability to effectively adapt various source

level analyses for executables.

Just as source-code symbolic analysis tools provide information about the symbolic values of programs variables, an executable-level symbolic analysis framework should provide information about the symbolic values of the contents of memory locations. We present a novel symbolic analysis framework for executables which computes a set of symbolic expressions, a *Symbolic Value Set*, that each data object (not just registers but also memory locations) can hold at each program point. The *Symbolic Value Set* is an abstraction for representing the possible values of each data object in terms of other program expressions.

This novel symbolic analysis framework has multiple applications. First, it improves the efficacy of various analysis like alias analysis and optimizations like redundancy elimination for executables. This results in a more aggressive optimization of executables. Second, it is useful for simplifying or speeding up subsequent binary analysis. For example, various bug testing tools employ advanced constraint solvers for detecting errors in a program. It has been shown that the time for various such decision procedures to return a satisfying answer for a query can be cut in half by using program optimization to simplify the query first [34]. Third, it improves the performance of various advanced transformations for executables. Recently, various researchers have suggested techniques for performing automatic parallelization from executables [162, 93]. None of these binary parallelization methods can currently apply advanced symbolic decisions like symbolic difference [76], which have been proved to be very effective in source-level parallelization methods. The proposed symbolic analysis framework will further improve the efficiency of all these paral-

lization efforts by exposing more data independences which cannot be captured by their existing methods.

Further, our framework does not use any symbolic, relocation, or debug information since these are usually absent in deployed executables.

4.2 Related Work

In this section, we discuss related work pertaining to (i) Symbolic analysis, (ii) Symbolic execution, (iii) Value numbering, and (iv) Binary Analysis.

Symbolic Analysis: There has been an extensive body of work employing symbolic analysis for analyzing and optimizing programs. Various techniques broadly differ in the symbolic abstraction which is maintained as part of their analysis. Cousot [52] proposed an early method of using abstract interpretation to discover the linear relationships between variables. Patterson [113] and Harrison [78] present methods for computing value ranges of program variables and employ it for improving static branch prediction [113]. Rugina et al [124] employ symbolic constraint solvers to determine the bounds of each variable in terms of its symbolic values at the entry point of the program. Padua et al [144] developed a system for computing symbolic values of expressions using a demand-driven backward substitution analysis on Gated-SSA form.

Symbolic analysis has been used extensively in the parallelization community to support the detection of parallelism and the optimization of programs. Haghghat et al [76](Parafrese-2) present a symbolic analysis framework for computing a closed

form expression of induction variables as well as for analyzing program properties that are essential in effective detection and exploitation of parallelization. Blume et al [31](Polaris) present a symbolic range propagation mechanism to determine the relationship between any two arbitrary symbolic expressions by maintaining a set of symbolic range constraints for each program variable. They further employ their symbolic ranges to improve data dependence queries. The SUIF compiler [77] employs symbolic analysis to represent array indices in a symbolic form of loop index variables to apply array dependence tests. Fahringer et al [67] present a unified symbolic evaluation framework, combining both data and control flow, for determining the symbolic expressions of variables as algebraic functions over program input data.

All the above methods are source-code symbolic analysis techniques and obtain symbolic expressions for only the variables. They lose a great deal of precision when applied to binary executables directly due to the presence of memory accesses. On the other hand, we present a symbolic analysis framework for executables which tracks memory locations as well, and does not lose precision in the presence of memory accesses.

Symbolic execution: There has been a great deal of work on symbolic execution in the field of model checking [39, 38, 127]. The only similarity between symbolic execution and symbolic analysis is that both use symbolic constraints to represent values, other than that, they are not very related. Our symbolic analysis is an abstract interpretation method which determines a set of symbolic expressions for each object. On the other hand, symbolic execution generates and maintains

symbolic constraints per program path and does not generalize constraints to all paths. Symbolic execution relies on constraint solvers to determine the feasibility of each path and is employed mainly for bug testing of programs.

Value numbering: Various algorithms have been suggested to discover the equivalence of expressions in a program. Since the equivalence problem is undecidable, compilers typically implement algorithms that solve a restricted problem of Herbrand equivalence. Most Global Value numbering algorithms are based on an early algorithm by Kildall [88], where equivalences are discovered using abstract interpretation on the lattice of Herbrand equivalences. Although the algorithm is precise, it has exponential cost in compile time. Later methods, including the algorithms by Alpern, Wegman and Zadeck (AWZ) [7] and Rosen, Wegman and Zadeck (RWZ) [123], suggest more efficient algorithms for discovering Herbrand equivalences based on the SSA form of the program. Gulwani et al [71, 72] present a random interpretation-based GVN algorithm that discovers as many Herbrand equivalences as the abstract interpretation algorithm of Kildall [88], while retaining the polynomial-time complexity of more efficient algorithms like AWZ [7]. VanDrunen et al [150] present a value-based partial redundancy algorithm, which effectively made value numbering a path sensitive algorithm. Bodik et al [32] combined value numbering with backward symbolic propagation and path sensitive data-flow analysis to propose a strong optimization framework.

However, all these value numbering algorithms are based on variables alone and none of these variables propagate value numbers across memory locations. Hence, these algorithms have limited application in the case of executables. In contrast,

our value numbering algorithm is implemented over a symbolic analysis framework which tracks symbolic values for memory locations as well, thereby exposing more equivalence in executables.

Binary analysis and optimization: There has been an extensive body of work on analyzing executables. The work that is closely related to our work are alias-analysis algorithms proposed by Debray et al [58], dependence analysis proposed by Amme [8] and Value Set Analysis method proposed by Gogul et al [21]. Debray [58] developed an alias-analysis algorithm for executables where the basic goal is to find an over-approximation of the set of values that each register can hold at each program point. Amme et al [8] also present a similar mechanism for deriving a set of values for each register but presented methods to avoid the loss of precision at program join points. However, the biggest limitation of both these methods is that they do not track memory locations and hence, lose a great deal of precision at each memory access.

Gogul et al [21] present Value Set Analysis that finds an over-approximation of the set of constant and memory address ranges that each abstract data object can hold at each program point. However symbolic analysis is a different problem from VSA - symbolic analysis derives symbolic expressions (rather than constants and memory address ranges) which each abstract data object can hold, enabling the applicability of our frameworks for detecting equivalences as well as for symbolic analysis to detect program parallelization. There have been other binary analysis tools like BitBlaze [136], Jakstab [89], BAP [36], UQBT [49] and none of them perform customized symbolic analysis for executables.

There are various binary analysis tools [121, 28, 95] which analyze executables in the presence of additional information like symbol tables or debugging information. Such information is usually absent in deployed executables and our methods do not make any assumption about the presence of such extra information. In addition, none of them deal with the problem of symbolic analysis.

Recently, there has been some amount of work on parallelizing executables. Kotha et al [93] present a method to automatically parallelize executables using a binary rewriter. They adapt source-level affine parallelization methods for executables. Yardimci and Franz [162] present non-affine automatic parallelism in a binary rewriter. Our symbolic analysis methods will further improve the efficiency of all these parallelization efforts by improving data dependence queries, thereby exposing more parallelism in programs.

4.3 Contribution

In this section, we discuss various analyses and optimizations which can be efficiently represented in our framework.

4.3.1 Redundancy elimination

The problem of determining the equivalence of two computations is undecidable in general. Consequently, compilers typically solve a restricted problem, where expressions are considered equivalent if and only if they are computed using the same operator applied on equivalent operands. This form of equivalence, where the

	Allocations: a: -4(%ebp) b:-8(%ebp) c: -12(%ebp) d: -16(%ebp)
b = a+2;	1 mov -4(%ebp), %eax //Load a
c = a + 12;	2 add \$2, %eax //Compute a+2
	3 mov %eax, -8(%ebp) //Store b
d = b + 10;	4 mov -4(%ebp), %eax //Load a
	5 add \$12, %eax //Compute a+12
	6 mov %eax, -12(%ebp) //Store c
Symbolic Relations:	
c = a+12	7 mov -8(%ebp), %eax //Load b
d = a+12	8 add \$10, %eax //Computer b+10
	9 mov %eax, -16(%ebp) //Store d
(a)	(b)

Figure 4.1: (a) A sample C code (b) Corresponding assembly code, the second operand in the instruction is the destination

operators are treated as uninterpreted functions, is called *Herbrand equivalence* [73].

Value numbering optimization determines when two computations in a program are Herbrand equivalent and eliminates one of them using a semantic preserving transformation. Various advanced redundancy elimination algorithms have been proposed which add semantic interpretation of various operators, thereby coupling symbolic analysis with the value numbering optimization; resulting in the discovery of more equivalent computations than defined by Herbrand equivalence [32].

However, all these techniques operate only on variables and treat memory accesses very conservatively. Although, they are effective in discovering equivalences in source code, not maintaining symbolic abstractions for memory locations renders them ineffective for discovering equivalences in the executables.

Fig 4.1(a) shows a small source code example and corresponding relations between various computations determined through symbolic analysis. The obtained symbolic relations expose the equivalence between the computations for variables

Program Point	Expression	Value Number
Line 1	x1	v1
Line 2	x1 + 2	v2
Line 4	x2	v3
Line 5	x2 + 12	v4
Line 7	x3	v5
Line 8	x3 + 10	v6

Program Point	Expression	Value Number
Line 1	tmp	v1
Line 2	tmp + 2	v2
Line 4	tmp	v1
Line 5	tmp + 12	v3
Line 7	tmp + 2	v2
Line 8	tmp + 12	v3

(a) (b)

Figure 4.2: (a) Value numbering obtained without propagation through memory locations (b) Value numbering with propagation through memory locations

c and d and existing symbolic analysis based value numbering methods [32] are sufficient in removing the redundant computation (for variable d).

Unfortunately, when source-level symbolic analysis methods are applied to executables, they cannot prove the equivalence between computations c and d in the example in Fig 4.1(a). Fig 4.1(b) shows a sample code which might arise when the code example in Fig 4.1(a) is converted to an executable. Fig 4.2(a) shows the symbolic relations and their corresponding value numbers when source-level symbolic analysis techniques are applied to the assembly code in Fig 4.1(b). Here, variables a , b , c and d are allocated to memory locations. Since symbolic analysis does not propagate symbolic expressions across memory locations, a new symbol is defined for each of the memory load instructions. The value numbers in Fig 4.2(a) depict that the equivalence of computations at Line 5 (variable c) and Line 8 (variable d) cannot be established.

The representation of symbolic abstraction for memory locations can eliminate this limitation as shown in Fig 4.2(b). Suppose, the variable a (memory location

$-4(\%ebp)$) has value tmp in the environment of symbolic abstraction. The representation of symbolic abstraction for memory locations implies that the variable $\%eax$ at Line 1 and Line 4 are assigned the value tmp in this environment. Similarly, the memory location $-8(\%ebp)$ at Line 3 and the variable $\%eax$ at Line 7 are assigned value $tmp+2$. Propagation of these symbolic values expose the equivalency between computations at Line 5 (variable c) and Line 8 (variable d).

The above example shows that maintaining symbolic abstraction for memory locations in executables has multiple advantages. It helps in more exposing more equivalent computations and it also results in a more effective redundancy elimination of memory access instructions. The value numbers in Fig 4.2(b) establish an equivalence between the load instructions at Line 2 and Line 7 and results in elimination of the latter, thereby improving a well studied optimization (*load redundancy elimination*) for executables [130].

4.3.2 Program Parallelization

Compilers employ various program analysis techniques to exploit concurrency on multiple processors. The notion of data dependence captures the most important properties of a program for efficient parallel execution on multicores and parallel machines. The dependence structure of a program defines the necessary constraints on the order of execution of program components. Various dependence analyses like array subscript analysis [157], distance vectors [25], integer programming based tests [118] and GCD tests [142, 26] have been suggested for determining the depen-

<pre> T2 = T1 + 1; for i = 0,N m = 2*i; j = m + T1; k = m + T2; A(j) = A(k) </pre> <p>Symbolic Relations $j = T1 + 2i$ $k = T1 + 2i + 1$</p>	<pre> A: A_mem m: 4[%esp], j: 8[%esp], k: 12[%esp], N: 16[%esp] i: eax, T1: ebx, T2: ecx 1 mov %ebx,%ecx 2 add 1,%ecx //T2 = T1+1 3 mov 0,%eax //Initializing i L1: 4 mov %eax, %edx //Calculating m 5 mul 2,%edx 6 mov %edx, 4[%esp] 7 mov 4[%esp], %edx //Calculating j 8 add %ebx,%edx 9 mov %edx, 8[%esp] 10 mov 4[%esp], %edx //Calculating k 11 add %ecx,%edx 12 mov %edx, 12[%esp] 13 mov 12(%esp), %edx //Array move 14 movl A_mem[%edx], %esi 15 mov 8[%esp], %edx 16 mov %esi, A_mem[%edx] 17 addl \$1, %eax //Increment i 18 cmpl %eax, 16(%esp) //compare N and i 19 jl L1 </pre>
(a)	(b)

Figure 4.3: (a) A sample C code (b) Corresponding assembly code, the second operand in the instruction is the destination

dence structure of a program and for determining parallel tasks.

However, these data dependence tests are more effective if the array subscript expressions are represented as affine expressions directly in terms of loop indices and loop invariants, rather than indirectly via other locations. As discussed in various parallelizing compilers like Parafrase [76], SUIF [77], Polaris [31], a large percentage of parallelization benchmarks have array references with symbolic terms other than loop induction variables and have symbolic loop bounds.

Symbolic analysis has been suggested as an important technique for improving

the data dependence decisions taken by a compiler in such scenarios. It is a very effective technique which represents array subscripts and loop bounds as a symbolic expression, describing its value in terms of constants, loop-invariant symbolic constants and loop indices. Standard dependence tests can then be employed to resolve data dependence queries [77, 31]. Advanced symbolic analysis based tests like symbolic difference [76] have also been proposed to determine the dependence structure when standard dependence tests fail due to the lack of information about certain variables at compile time. Symbolic analysis also enables various transformations like induction variable substitution and array privatization which further aid in exposing the dependence structure of a program [76, 77, 31].

Fig 4.3(a) shows a small loop example where symbolic analysis is imperative for establishing the absence of a loop carried dependency. The symbolic relations in Fig 4.3(a) (obtained by applying symbolic analysis) show that access $A(j)$ is equivalent to $A(T1+2*i)$ whereas access $A(k)$ is equivalent to $A(T1+2*i+1)$. Various data dependence tests on these array reference expressions can reveal that these accesses will always refer to a disjoint set of locations (if $T1$ is even, then $T1+2*i$ will always have even values; whereas $T1+2*i+1$ will have odd values, and vice versa). Consequently, the loop in Fig 4.3(a) is determined to be parallelizable from source code.

Unfortunately, source-level symbolic analysis might not be able to obtain such affine expressions for array subscripts from an executable. Fig 4.3(b) displays a possible assembly code version of the loop in Fig 4.3(a). Here, variables m , j and k are allocated to memory locations. As mentioned before, existing symbolic analyses

Symbolic Relations:	Symbolic Relations:
Line5 (%edx) (m) = $2 * \%eax$	Line5 (%edx) (m) = $2 * \%eax$
Line 7(%edx) = x1	Line 7(%edx) = $2 * \%eax$
Line8 (%edx) = $x1 + \%ebx$	Line8 (%edx) = $2 * \%eax + \%ebx$
Line 10 %edx = x2	Line 10 %edx = $2 * \%eax$
Line11 (%edx) = $x2 + \%ebx + 1$	Line11 (%edx) = $2 * \%eax + \%ebx + 1$
Line 13(%edx) = x3	Line 13(%edx) = $2 * \%eax + \%ebx + 1$
Line14(%edx) (k) = x3	Line14(%edx) (k) = $2 * \%eax + \%ebx + 1$
Line 15(%edx) = x4	Line 15(%edx) = $2 * \%eax + \%ebx$
Line16(%edx) (j) = x4	Line16(%edx) (j) = $2 * \%eax + \%ebx$

(a)
(b)

Figure 4.4: (a) Symbolic expressions obtained with no memory propagations (b) Symbolic expressions with memory propagation

maintain symbolic abstractions only for variables. Hence, new symbols are created to represent each of the loaded values in the environment of symbolic abstraction. Symbolic expressions in Fig 4.4(a) corresponding to the assembly code in Fig 4.3(b) depict that no determinable relation can be obtained between variables j (%edx at Line 16) and k (%edx at Line 14). Consequently, data dependence analysis conservatively assumes the presence of a loop carried dependence, which limits the parallelizability of this loop.

On the other hand, maintaining symbolic abstractions for underlying memory locations enables the discovery of such affine expressions from executables also. Fig 4.4(b) shows the obtained symbolic expressions when abstractions are also maintained for memory locations. The symbolic expressions for j (%edx at Line 14) and k (%edx at Line 16) are affine expressions ($2 * \%eax + \%ebx$ and $2 * \%eax + \%ebx + 1$ respectively) in terms of loop indices and invariants. Consequently, standard data dependence can reveal the lack of a loop carried dependence resulting in parallelization of this loop.

There has been a recent surge in research methods exploring parallelization of executables [93, 162]. However, executables-level parallelization is still in infancy stage as compared to source-level parallelization. Our framework will enable the application of an important source-level analysis framework to the executables, thereby improving the data dependence decisions capability of all such executable level parallelization techniques.

4.3.3 Alias analysis

Alias analysis has been extensively studied for source code. Recently, there has been a surge of interest in extending pointer and alias analysis techniques to low-level code. Early alias analysis techniques by Debray [58] and Amme et al [8] maintained internal abstraction for only the variables. Consequently, they lost a great deal of precision at memory accesses. Gogul et al [21] presented a novel Value Set Analysis (VSA) framework which eliminated this limitation. VSA is a combined numeric and pointer analysis which determines an over-approximation of the set of memory addresses as well as the set of integer values that each data object (a register or a memory location) can hold at each program point.

Although VSA is a very powerful alias analysis framework for executables, the symbolic abstraction, as maintained in our technique, can aid the VSA abstraction in resolving aliasing queries in some scenarios. Fig 4.5 depicts an example of such a scenario. In Fig 4.5, suppose the variable `%ebx` at Line 1 has value \top in the VSA abstraction. This represents the fact that VSA could not narrow down the possible

1.	mov %ebx, 8(%esp)	//mov %ebx to 8[%esp]
2.	mov (%ebx), %ecx	//load from memory location //pointed to by %ebx
3.	mov 8(%esp), %edx	//load from 8[%esp]
4.	mov 4(%edx), %eax	//load from memory location //pointed to by (%edx+4)

Figure 4.5: A sample assembly code, second operand in the instruction is the destination

set of values of $\%ebx$; hence it is the universal set (\top). The VSA abstraction for the memory location $8[\%esp]$ (and variable $\%edx$) at Line 3 also has value \top . Consequently, the alias relation between memory accesses at Line 2 and Line 4 can only be established as *may-alias*, since $\top + 4 \equiv \top$ in the VSA abstraction.

However, this result can be improved through our symbolic analysis framework, where we maintain symbolic value sets corresponding to each data object. In Fig 4.5, suppose the variable $\%ebx$ is defined to have value sym ($\neq \top$) in the environment of symbolic abstraction. The representation of symbolic abstraction for the memory location $8[\%esp]$ results in the variable $\%edx$ at Line 3 also having value sym . Comparing the memory locations at Line 2 and Line 4 in the symbolic abstract environment reveals that these two instructions access distinct memory locations, since $sym \neq sym + 4$. Consequently, the alias relation can be established as *no-alias* instead of *may-alias* in previous case.

We do not envision our technique as a replacement to existing alias analysis

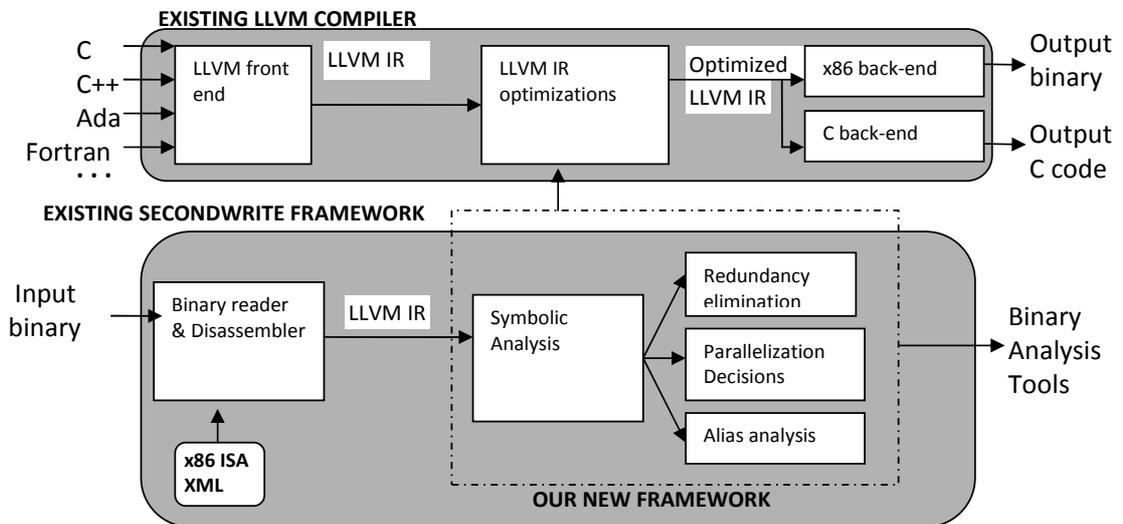


Figure 4.6: *Organization of the system*

frameworks; instead we view it as an additional abstract environment for solving aliasing queries. This is driven by a simple observation that alias analyses are composable. Multiple analysis techniques can be combined to yield a better overall analysis than any of its components. An aliasing query involving two memory references can be resolved if any of the multiple alias analysis methods can resolve the query. The VSA abstraction, combined with the symbolic abstraction, provides a stronger alias analysis framework for executables than any existing aliasing framework.

4.4 Overview

Fig 4.6 presents an overview of the our binary analysis framework. Our framework is built over existing SecondWrite framework as presented in [93, 111]. SecondWrite translates the input x86 binary code to the intermediate format of the LLVM Com-

piler [96]. LLVM, which stands for Low-Level Virtual Machine, is a well-known, open-source compiler developed at the University of Illinois; it is now maintained by Apple Inc. *This conversion back to a compiler intermediate representation (IR) is not a necessity for the work we present; any binary system can use our analysis.* However, using LLVM IR enables us to use LLVM's rich infrastructure, such as control-flow analysis, dataflow analysis, and optimization passes, so that we did not have to write our own for the system. LLVM IR obtained above can be passed through our analysis system to obtain an optimized IR which can be passed to further binary analysis tools. In addition, LLVM's x86 code generator can be used to obtain a rewritten binary.

SecondWrite implements various mechanisms to obtain an intermediate representation which contains features like procedure arguments, return values, types and high-level control flow. SecondWrite also employs extra mechanisms to safely handle indirect calls and indirect branches [135]. It employs alias analysis frameworks present in LLVM to discover all the possible target procedures at an indirect call-site, given by the points-to set of the operand in indirect call instruction. An edge is added from the indirect call-site to all its possible target procedures. Indirect branches are mostly present due to jump tables in the binary. Procedure boundary determination techniques are devised to limit the possible branch targets within the current procedure and extra control flow edges are added corresponding to the possible targets determined by alias analysis. If one of the target is outside procedure boundary, it is handled as an indirect call.

4.4.1 Memory abstraction

There are two prerequisites for implementing a symbolic analysis for executables that can track symbolic information for memory locations. First, a memory abstraction is needed to represent a large number of runtime addresses by a smaller and finite set of abstract locations. Second, executables regularly employ the *indirect-addressing* mode for accessing memory locations.¹ A mechanism is needed to determine the set of memory locations which can be accessed by any *direct* or *indirect* memory access instruction.

We employ the concept of abstract memory regions and abstract locations (*a-locs*), defined by *Value Set Analysis* (VSA) [21, 22], to build a memory abstraction for symbolic analysis. The address space of a program is divided into several non-overlapping memory regions. For a program, the set of memory regions consists of one abstract region per procedure for its stack frame, one abstract region per heap allocation and a global region. Each memory region is further abstracted through a set of *a-locs*. Intuitively, *a-locs* correspond to program variables in each memory region. An *a-loc* is characterized by two attributes: its relative offset in the region with respect to other *a-locs* and its size.

Having defined *a-locs* as above, VSA computes an over-approximation of the set of integers and the set of memory addresses (collectively referred to as a *value-set*) that each register and each *a-loc* holds at a particular program point. VSA

¹For our purposes, a memory reference uses direct addressing if the address being accessed is a constant that is part of the instruction, or has a constant offset from the stack pointer. Otherwise it uses an indirect addressing mode, in which case the location being accessed is statically unknown.

$Sym := Sym + T T$ $T := T * F F$ $F := l n$ $l := [IR \text{ Variables}]$ $n := [Int]$

Figure 4.7: Grammar for symbolic expressions. $+$ and $*$ are standard arithmetic operators, Int is the set of all integers, IR Variables are symbols in the obtained intermediate representation

employs advanced affine relation analysis and loop bound analysis to conservatively bound the memory locations accessed by any instruction. Hence, this algorithm can be used to determine the set of all possible memory locations referred to by all the *direct* and *indirect* memory access instructions. More details about this algorithm can be found in [21].

4.5 Symbolic Abstraction

There are a variety of choices to represent the abstraction for symbolic domain values. Fig 4.7 presents the grammar for representing the symbolic expressions in our abstraction. As evident from Fig 4.7, symbolic expressions are numeric algebraic polynomials containing sums of product terms of variables.

Symbolic Value Set: The objects in our abstract symbolic domain are *Symbolic Value Sets* – a finite set of canonical symbolic expressions defined by the Grammar in Fig 4.7. A *Symbolic Value Set* represents a conservative over-approximation of the the set of symbolic values that each data object (IR variables and *a-locs*) holds at a particular program point.

Various operations are defined on this symbolic expression as described below.

In general, *sym* terms below refer to symbolic expressions, not individual symbols.

(a) *Create Symbolic Expression* : $Sym(var)$:

Associates a new symbolic expression with a program variable *var*, in the symbolic abstraction domain.

(b) *Canonicalize Operator*: $Can(sym)$:

Rearranges the symbolic terms present in the symbolic expression *sym* in a unique canonical form (such as a lexicographical order determined by the pointers of variables in IR).

(c) *Addition Operator*: $sym1 + sym2$:

Computes a symbolic expression by adding *sym2* to *sym1* and returns the canonicalized version of the result.

(d) *Multiply Operator*: $sym1 * sym2$:

Computes a symbolic expression by applying the arithmetic multiplication operator between *sym1* and *sym2* and returns the canonicalized version of the result.

Indirect memory references in executables can update any memory location which aliases with the address of the indirect reference. *Symbolic Value Set* abstraction, which contains a set of symbolic expressions, is sufficient to represent

the possible initializations at multiple locations. In order to limit the exponential growth of symbolic expressions, we employ a limit on the cardinality of symbolic value set, at the cost of some precision. The following operations are defined on symbolic value sets:

(a) *Union Operation:* $SymValSet_1 \cup SymValSet_2$

This operator computes the join of two symbolic value sets $SymValSet_1$ and $SymValSet_2$

(b) *Add Operator:* $SymValSet_1 \oplus SymValSet_2$

This operator computes a new symbolic value set by adding each symbolic expression present in $SymValSet_2$ to each symbolic expression present in $SymValSet_1$.

Mathematically, this operation can be represented as

$$\begin{aligned}
 SymValSet_1 \oplus SymValSet_2 = \{sym1 + sym2 \mid \\
 sym1 \in SymValSet_1, \\
 sym2 \in SymValSet_2\}
 \end{aligned} \tag{4.1}$$

(c) *Multiply Operator:* $SymValSet_1 \otimes SymValSet_2$

This operator computes a new symbolic value set by applying the multiplication operator between each symbolic expression present in $SymValSet_1$ and $SymValSet_2$.

Mathematically, this operation can be represented as

$$\begin{aligned}
SymValSet_1 \otimes SymValSet_2 &= \{sym1 * sym2 \mid \\
&sym1 \in SymValSet_1, \\
&sym2 \in SymValSet_2\}
\end{aligned} \tag{4.2}$$

(d) *Widen*: $\nabla SymValSet_1$

This operations implements the inherent widening operation in our symbolic abstraction environment. As mentioned above, the abstract symbolic domain has infinite ascending chains. In order to limit the exponential growth of symbolic expressions, widening needs to be implemented at some nodes of the analysis. If the required cardinality increases beyond a limit, we invalidate the current symbolic value set.

$$\begin{aligned}
\nabla SymValSet_1 &= \{if \ |SymValSet_1| > LIMIT, \\
&then \top \\
&else \ SymValSet_1\}
\end{aligned} \tag{4.3}$$

4.6 Symbolic Value Analysis

This section describes the Symbolic Value Analysis. Symbolic Value Analysis is a flow-sensitive, context insensitive analysis which computes a conservative over-approximation of a set of values that each data object (variables and *a-locs*) can hold at each program point. The values are represented in an abstract symbolic domain presented in Section 4.5.

4.6.1 Intraprocedural Analysis

This subsection describes the intraprocedural version of symbolic value analysis. The inter-procedural version will be described in the next subsection. Symbolic value analysis is defined as an abstract interpretation over the control flow graph of a procedure. Symbolic value analysis effectively computes a *Symbolic Map* at each program point, which is a representation of a mapping between the data objects and corresponding symbolic value sets.

Our method assumes that the symbols corresponding to the binary code's registers have been converted to single-static assignment (SSA) form in the binary tool's intermediate representation (IR) before running our analysis. SSA form is widely used in many compilers and binary analyzers, including SecondWrite, for doing data flow analysis. Since in SSA form each variable is assigned exactly once, a single symbolic map is sufficient to maintain flow-sensitive symbolic value sets for variables. However, memory locations are usually not implemented in SSA format in IR. Consequently, a symbolic map is maintained at each program point to repre-

sent flow-sensitive symbolic value sets for memory locations. Hence, symbolic value analysis effectively computes the following symbolic maps:

SR : Map between variables (corresponding to registers in the input binary, and variables in the IR) and their corresponding symbolic value sets

SM_e : Map between a -locs and their corresponding symbolic value sets before a program point e

Similar to any data-flow analysis, symbolic value analysis is applied iteratively by traversing the CFG of a procedure in a topological order. The symbolic maps SR and SM_e are initialized as empty sets at the beginning of the analysis. The iteration is continued until the maps reach a fixed point.

A symbolic map may contain at most one entry for each distinct data object. A lookup in the map SR corresponding to a variable var not in the map, results in a single entry symbolic value set containing a new symbolic expression $Sym(var)$. Correspondingly, a lookup for an a -loc not in the map SM_e returns \top .

The algorithm is implemented on the intermediate representation of the executable, but we present our algorithm on C-like pseudo instructions for ease of understanding. Each instruction in the intermediate representation implements a transfer function which translates the symbolic maps defined at its input to the symbolic maps at its output.

The following definitions are introduced to ease the presentation.

R_i : IR (SSA) variables

e : A program point

SM'_e : Map between *a-locs* and their corresponding symbolic value sets after program point e

$SR(r)$: Mapping of data object r (variable) in map SR

$SM_e(r)$: Mapping of data object r (*a-loc*) in map SM_e

$VS_e(r)$: Set of memory addresses that data object r (variable or *a-loc*) can hold at a program point e (obtained by Value Set Analysis)

(r, SV) : Pairing between a data object r and a symbolic value set SV

VSA includes a concept of *fully accessed* and *partially accessed a-locs*. In order to understand partial *a-locs*, consider that value set of a particular data object r at a program point e , $VS_e(r)$, contains a list of memory addresses that the data object r can hold at current program point e . If this object is dereferenced in a memory access instruction of size s , the *a-locs*, that are of size s and whose starting addresses are in set $VS_e(r)$, represents the fully accessed *a-locs*. The partially accessed *a-locs* consists (i) *a-locs* whose starting addresses are in $VS_e(r)$ but are not of size s and (ii) *a-locs* whose addresses are in $VS_e(r)$ but whose starting addresses and size do not meet the condition to be fully accessed *a-locs*. As per the notation in VSA [21], this operation is mathematically represented as:

$$\{F, P\} = *(VS_e(r), s)$$

In above representation, F represents the fully accessed *a-locs* and P represent the partially accessed *a-locs*. As the name suggests, only some portion of a partial *a-loc* is updated or referenced in a memory access instruction. Partially accessed *a-locs* are problematic since their symbolic expressions are hard to derive after they are written to; hence, they are treated conservatively in our analysis, as will be explained below.

Table 4.1 shows the mathematical forms of transfer functions for each instruction. Each row in this table represents the transfer function corresponding to an instruction. Below, each of these transfer functions is discussed in detail

1. *Assignment: $e : R1 := R2$*

This is the basic operation of symbolic analysis where symbolic analysis behaves similarly to the concrete evaluation. As presented in Row 1 in Table 4.1, any existing entry in the symbolic map *SR* corresponding to the variable *R1* (computed in an earlier iteration) is removed from the map and the symbolic value set of variable *R2* is assigned to variable *R1*.

2. *Arithmetic Operation: $e : R3 := R2 OP R1$*

In such scenarios, the symbolic value analysis evaluates the symbolic values according to the underlying mathematical operator. The evaluation is defined for addition,

Name	Operation	Transfer Function
1. Assignment	$e : R1 := R2$	$SR = \{SR - SR(R1)\} \cup \{(R1, SR(R2))\}$
2. Arithmetic Operation	$e : R3 := R2 OP R1$	$if OP = +$ $tmp = \nabla(SR(R2) \oplus SR(R1))$ $if OP = *$ $tmp = \nabla(SR(R2) \otimes SR(R1))$ $else$ <i>//Create a new symbolic expression</i> $tmp = Sym(R3)$ $SR = \{SR - SR(R3)\} \cup \{(R3, tmp)\}$
3. Memory Load	$e : R1 := *(R2)$	$\{F, P\} = *(VS_e(R2), s)$ $if P = 0$ $tmp = \nabla(\bigcup_{v \in F} SM_e(v))$ $else$ $tmp = \top$ $SR = \{SR - SR(R1)\} \cup \{(R1, tmp)\}$
4. Memory Store	$e : *(R2) := R1$	$\{F, P\} = *(VS_e(R2), s)$ $if F = 1 \ \& \ P = 0 \ \&$ $Func \ is \ not \ recursive \ \&$ $F \ has \ no \ heap \ a-locs$ <i>//Strong Update</i> $SM'_e = \{\{SM_e - SM_e(v)\} \cup \{(v, SR(R1))\}$ $\quad \quad \quad v \in F\}$ $else$ <i>//Weak Update</i> $SM'_e = \{\{SM_e - SM_e(y) \mid y \in \{F \cup P\}\} \cup$ $\{(v, \nabla(SR(R1) \cup SM_e(v))) \mid v \in F\} \cup$ $\{(p, \top) \mid p \in P\}\}$
5. SSA Phi Function	$e : R_{n+1} = \phi(R_1, R_2, \dots, R_n)$	$SR = \{SR - SR(R_{n+1})\} \cup \{(R1, \nabla(\bigcup_{i \in (1, n)} SR(R_i)))\}$

Table 4.1: Transfer functions for each instruction in a procedure *Func*. Here, s denotes the size of dereference in a memory access instruction.

subtraction and multiplication operators. In case of addition, the underlying $Add(\oplus)$ operator is employed and the underlying $Multiplication(\otimes)$ operator is employed for evaluating the values in the case of multiplication operation. Subtraction operation is handled analogous to the addition operation by reversing the sign of each coefficient in the symbolic expressions of second operand, $R1$. Hence, we only mention addition operation to simplify the presentation. Since the remaining arithmetic and logical operations are not represented, a new symbolic expression is introduced to represent the result of the computation as presented in Row 2 in Table 4.1.

The introduction of a new symbolic expression is governed by a balance between the precision and analysis cost. The canonical symbolic expression term needs to include other arithmetic and logical expressions to represent the remaining operations. The current canonical expression is chosen to limit the analysis cost of extra operations.

3. Memory Load $e : R1 := *(R2)$

The propagation of symbolic values in memory loads relies on employing the underlying Value Set Analysis. VSA provides a set of fully accessed and partially accessed $a\text{-locs}$ that the object $R2$ can hold at current program point e corresponding to current dereference size s . If the current memory instruction does not access any partial $a\text{-loc}$, the symbolic value of variable $R1$ is computed by unioning the symbolic values corresponding to each of the possible $a\text{-loc}$. Otherwise, it is assigned \top .

4. *Memory store* $e : *(R2) := R1$

The propagation of symbolic values during memory stores also employs Value set analysis. The propagation of symbolic values is governed by current memory store accessing a single *a-loc* or multiple *a-locs*. If the current memory store only updates a single fully accessed *a-loc* (referred to as a strong update), the existing symbolic values of the destination memory location is replaced by the symbolic set. The memory stores which update a partial *a-loc* or update multiple *a-locs* are referred to as weak updates. In such cases, the new symbolic values are unioned with the existing ones to obtain the updated symbolic value set of fully accessed *a-locs*. The partially accessed *a-locs* are assigned symbolic \top .

As explained in VSA [21], memory region corresponding to the stack frame of a recursive procedure or corresponding to heap allocations potentially represent more than one concrete *a-loc*. Hence, the assignments to their *a-locs* are also modeled by weak updates.

5. *SSA Phi Function*: $e : R_{n+1} = \phi(R_1, R_2, \dots, R_n)$ At join points in the control flow of a procedure, the symbolic value sets from all the predecessors are unioned to obtain a new symbolic value set.

As per any flow-sensitive data-flow analysis, the symbolic map at a join point in control flow graph is determined by unioning the symbolic maps from all the predecessors. Existing symbolic frameworks have a property that every variable has only one abstract symbolic value at a certain program point [76]. A new abstract value is created for a variable at a join point if the variable has different abstract

symbolic values on different incoming edges. In contrast, our framework avoids this loss of precision at join points by unioning the abstract values from all the paths to obtain a symbolic value set for each data object. This increased precision results in a more precise dependence analysis and in more effective resolution of aliasing queries.

4.6.2 Interprocedural propagation

This subsection describes the interprocedural aspect of symbolic value analysis. Interprocedural analysis requires the correct handling of symbolic values at callsites and return points.

Existing binary analysis tools implement various methods to recognize procedure arguments and procedure returns independent of the calling conventions [164, 21, 111]. Various advanced data flow analysis have been suggested to recognize register arguments, register returns and stack based arguments. SecondWrite also implements various analyses to recognize the arguments. Once the arguments are recognized, an intermediate representation is formed where formal arguments and procedure returns are represented as a part of procedure definition and actual arguments and actual returns are explicitly represented as a part of a call instruction in the IR.

The symbolic value set of a formal argument for a procedure P is computed by unioning the symbolic value sets of corresponding actual arguments across all the call-sites for procedure P . Since binary programs always contain the entire program,

such whole-program analysis is always possible. Mathematically, the initialization of formal f_i of procedure P , where a_{ci} represents the corresponding actual argument at a callsite c , is represented as

$$SR = \{SR - SR(f_i)\} \cup \{(f_i, \nabla(\bigcup_{\forall c \in CallSites(P)} SR(a_{ci})))\} \quad (4.4)$$

In order to propagate the symbolic values of memory locations, the memory symbolic maps from each call site need to be unioned to determine the symbolic map at entry point P_{entry} of a procedure P , similar to the symbolic map propagation at join point in CFG of a procedure.

$$SM_{P_{entry}} = \bigcup_{\forall c \in CallSites(P)} SM_c \quad (4.5)$$

Similarly, at a return site, symbolic value set of return variable is evaluated from the internal symbolic map for variables. Symbolic map, just after a call instruction C , is computed by unioning the symbolic maps at all the return points in the called procedure P .

$$SM_C = \bigcup_{\forall r \in ReturnSites(P)} SM_r \quad (4.6)$$

Since VSA is an interprocedural analysis, it implicitly results in correct interprocedural propagation of symbolic values of underlying memory location. At each memory load or store instruction, VSA provides a set of possible *a-locs* (belonging to any procedure) which can be accessed by this instruction. The initialization of symbolic maps at the entry point of a callee procedure ensures that all the required symbolic values are propagated from the caller procedures to the callee procedure and are available at memory loads. Similarly, the join of symbolic maps at exit point in the caller procedures propagates the symbolic values modification from the callee procedure back to the caller procedures.

As presented in Section 4.4, we employ the alias analysis frameworks present in LLVM to discover all the possible target procedures at an indirect call-site and insert an edge from the indirect call-site to all its possible target procedures in the IR. This representation ensures that the interprocedural propagation as presented above is sufficient to propagate the information correctly at indirect callsites. The union in equation 4.6 is computed across all these possible target procedures P .

The externally called procedures are handled in one of the following three ways. First, procedures which are known not to affect the memory regions (e.g. `puts`, `sin`) are modeled as identity transformers (a NOP). External procedures like `malloc`, which create a memory region, are also modeled as identity transformers since we already handle these procedures by defining a memory abstraction `HeapRgn` corresponding to each allocation site. External procedures like `free`, which destroy a memory region, are conservatively modeled as NOP by our analysis. Next, unsafe but known external procedures (e.g. `memcpy`) are handled by widening the symbolic

value set of all `a-locs` in the memory regions possibly accessed by the procedure. Unknown external procedures (which include user defined libraries) are handled by widening the symbolic value set of registers and all `a-locs` in all the memory regions.

Recursive procedures: The analysis presented in Table 4.1 handles stores in recursive procedures as weak updates. However, in some cases default propagation of symbolic abstraction interprocedurally for recursive procedures might result in indefinite ascending chains. The widening operator in our analysis implicitly implements a fixed-point algorithm and prevents such exponential explosion of symbolic expressions.

4.7 Dependence Analysis

The effectiveness of parallelizing compilers is highly dependent on the accuracy and the preciseness of data dependence for array references in loop nests. As explained in Section 4.3, the dependence tests for parallelization require a closed form (affine) expression for array indices [76] in terms of loop index variables. The symbolic analysis presented in Section 4.6 discovers such affine expressions for array indices from executables, even if some of these indices are allocated to memory locations.

The widening operator in the symbolic abstraction might result in some loop index variables to have value \top . In order to obtain a closed form expression for such loop index variables and for the array indices which are based on these variables, symbolic analysis is applied on the loop body for two consecutive iterations resulting in a system of recurrence relations, similar to the method suggested by [76]. These

recurrent relations are solved to obtain value of expressions at different loop iterations as a function of loop index variables. For example, a variable i with initial value 0 and which is incremented by value 2 in each loop iteration is identified as a recurrence relation $\{0, 2, +\}$.

Existing source-level symbolic frameworks [76] obtain these recurrence relations for only the variables while our framework will obtain this recurrence relations for IR variables as well as for *a-locs*. Since a loop-index variable might be allocated to a memory location in an executable, our framework recognizes recurrence expressions for memory-allocated loop index variables also, which cannot be recognized by applying existing source-level frameworks to executables.

Existing parallelizing compilers based on symbolic analysis frameworks [76, 31, 77] collect the required information (recurrence relations and affine expressions) by symbolic analysis and perform dependence testing using a variety of techniques. Various common dependence techniques, as presented in [69, 77], employ recurrence relations and affine expressions between array indices to characterize the dependence structure in two aspects. First, they try to disprove the loop carried dependence between pairs of subscripted references to the same array variable. Second, if dependence exists, they try to characterize the dependence by determining the actual distance in terms of number of loop iterations (referred to as distance vector) between two accesses to the same memory location.

Since our symbolic value set contains multiple symbolic values, instead of a single abstract symbolic value, it entails some modifications to both these aspects of dependence tests. First, instead of testing the existence of dependence between two

references using their unique abstract symbolic value, the dependence tests need to be performed for each pair of abstract symbolic values belonging to the symbolic sets. Two references are considered dependent if the dependence exists for even one pair of abstract symbolic values. Mathematically, if S_1 and S_2 denote the symbolic value set of data objects corresponding to two references $d1$ and $d2$, the test for existence of dependence can be represented as:

$$Dependence(d1, d2) = \bigvee_{e1 \in S_1, e2 \in S_2} Dependence(e1, e2) \quad (4.7)$$

Next, if the above dependence tests fails to disprove the dependence, then for loop-carried dependences the distance vector, $DistVec$, is calculated by unioning the distance vectors determined from each pairs of abstract values.

$$DistVec(d1, d2) = \bigcup_{e1 \in S_1, e2 \in S_2} \{DistVec(e1, e2)\} \quad (4.8)$$

Traditional source-level frameworks represent a single distance vector for two accesses while the above test gives a union of distance vectors for any two accesses. Multiple distance vectors arise when array are referenced through pointer accesses, which can arise in an executable. A parallelization technique can define its own operation to combine this union of distance vector for determining parallel tasks.

```

1 Input: Control flow graph G
2 Output:
3 ValNum: Map between an assignment instruction I, where  $I \in G$ , and its value
   number
4 Local Definitions:
5 SymHash: Map between a symbolic value set and its value number
6 SymValSet(X): Symbolic value set of a variable X
7 CurValueNum = 0
8 Instruction  $I : X = (Operation, Operands)$ 
9 for each instruction I in reverse post-order traversal of G do
10 |   CurExpr = SymValSet(X)
11 |   if SymHash.HasEntry(CurExpr) then
12 |     Temp = SymHash[CurExpr]
13 |   else
14 |     Temp = CurValueNum;
15 |     CurValueNum++;
16 |     SymHash[CurExpr] = Temp
17 |   ValNum[I] = Temp

```

Algorithm 2: Value numbering on symbolic expressions

4.8 Value Numbering

As explained through an example in Fig 4.1 in Section 4.3, value numbering on memory based symbolic analysis frameworks exposes more equivalences than defined by traditional GVN algorithms. In this section, we present the details of our value numbering algorithm for recognizing equivalent computations.

Symbolic Value Analysis, as presented in Section 4.6, computes a set of symbolic expressions that each data object can hold at each program point. In order to employ this analysis for removing redundant computations, an abstract interpretation based algorithm is implemented on the lattice of symbolic expressions. This is in contrast with Kildall’s value numbering algorithms [88] which expose equivalence by implementing an abstract interpretation on the lattice of Herbrand equivalences.

Algorithm 2 describes our algorithm for determining the value numbers. The

process mimics the pessimistic version of the GVN algorithm presented in [123], although the optimistic value numbering algorithm can also be adopted [7]. Algorithm 2 improves over traditional GVN in three aspects. First, the equivalence relation is determined on underlying symbolic expressions rather than program variables, consequently, our value numbering discovers more equivalences than GVN where variable assignment is the only algebraic simplification. Second, memory load instructions are also considered for discovering equivalent computations whereas traditional GVN only considers arithmetic and logical assignment operations of the form $X = f(A, B)$ to determine the equivalence. Third, the propagation of symbolic expressions among underlying memory locations results in a more precise flow of symbolic values.

The presence of non-singleton symbolic value sets for the computations might complicate the discovery of equivalent computations in some scenarios. If two computations have the same symbolic value set of cardinality greater than one, then the representation cannot establish their equivalence. This is the case even if they take the same value in reality.

Fig 4.8 shows a small example exhibiting this problem. In this example, variable eax and ebx are Herbrand equivalent, reflected by the traditional value numbering in Fig 4.8 (right side of block $B3$). However, without improvement, the representation so far of our symbolic analysis (which incorporates memory locations) will not be able to prove this equivalence. According to the symbolic value propagation mechanism as presented in Section 4.6, the symbolic value set of the memory location ($8[esp]$) at the entry point of basic block $B3$ is a two element set

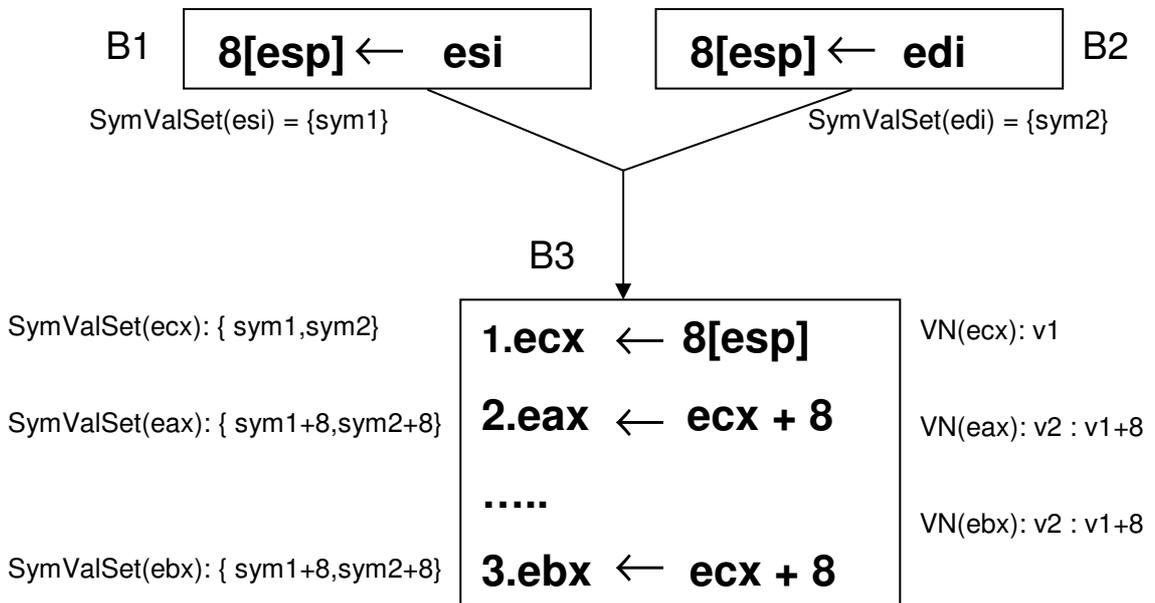


Figure 4.8: An example CFG showing the limitations of symbolic expressions for value numbering

$\{sym1, sym2\}$. Consequently, the symbolic value sets of computation 2 and 3 in basic block $B3$ are also of cardinality two with elements $\{sym1+8, sym2+8\}$. These two computations cannot be considered equivalent in the environment of symbolic abstraction since it is not possible to statically prove that these computations would refer to the same symbolic expression at runtime.

In order to discover all such equivalences while maintaining the inherent advantages of symbolic value sets, a new kind of operation is introduced to represent the memory loads. If the symbolic value set of a memory load, M_i , has cardinality greater than one, a new operator ϕ_{M_i} is introduced to represent this operation. Fig 4.9 shows the introduction of this operator for the example in Fig 4.8. This operator behaves as an uninterpreted operator in the symbolic analysis framework. Being an uninterpreted operator, a new symbolic expression is defined to represent

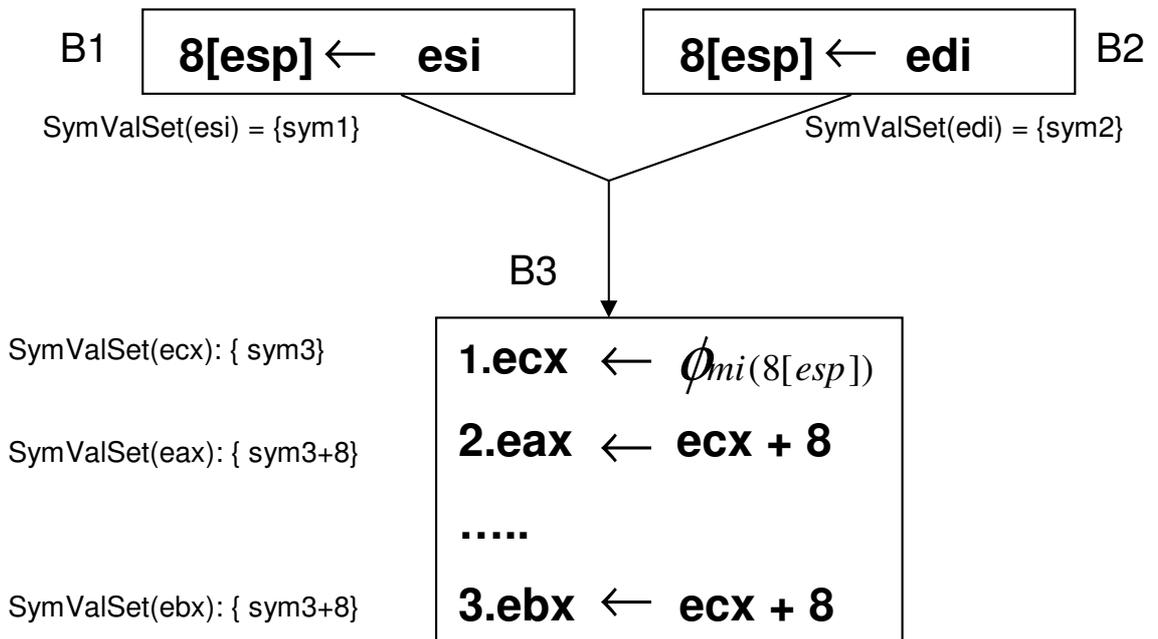


Figure 4.9: *Introduction of Phi for removing the limitations of symbolic expressions for value numbering*

the result of this computation. As shown in Fig 4.9, the presence of this uninterpreted operator as part of symbolic analysis exposes the Herbrand equivalence.

The introduction of this new operator ϕ_{M_i} ensures that the symbolic analysis framework retains its inherent advantages of exposing more equivalences due to the tracking of memory locations and semantic interpretation of operators and also discovers all the Herbrand equivalences which can be discovered by traditional value numbering.

As explained in Section 4.7, a symbolic value set is more precise than a single symbolic expression for data dependence analysis. In order to avoid losing this advantage, symbolic maps still hold a mapping between these uninterpreted operators and their symbolic value sets. For data dependency tests, the symbolic values of ϕ_{M_i} are recursively evaluated until we get rid of all such ϕ_{M_i} operators. This in-

ternal mapping and demand driven evaluation of symbolic value sets ensures that introduction of this new uninterpreted operator does not adversely impact the data dependence decisions.

Similar problem also arises for variables at join points (regular SSA *phi*). In such scenarios, we employ the mechanism proposed by [7] to represent *phi* as an uninterpreted operator.

4.9 Results

The symbolic analysis framework is implemented on LLVM IR as part of the Second-Write framework presented in Section 4.4. The evaluation is performed on several benchmarks from the SPEC2006 and OMP2001 suites and a real world program (apache server), as listed in Table 4.2. Benchmarks are compiled with gcc v4.3.1 with O3 flags (Full optimization) and results are obtained on a 2.4GHz 8-core Intel Nehalem machine running Ubuntu.

4.9.1 Static characteristics

Table 4.2 shows the running time and storage requirements of our symbolic analysis framework on various benchmarks. The numerical value of *Limit*, the maximum size of a symbolic value set, was kept to 5 in these experiments. The analysis time and the required storage is largely a function of number of procedures in the benchmark. The analysis time is typically low, within one minute, for most of the benchmarks except for some intensive benchmarks like gcc and dealII.

Application	Source	Lang	LOC	# Proc	Time(s)	Mem (MB)
bwaves	Spec2006	F	715	22	4.25	24.47
lbm	Spec2006	C	939	30	0.8	1.03
quake	OMP2001	C	1607	25	0.64	3.62
mcf	Spec2006	C	1695	36	0.31	2.85
art	OMP2001	C	1914	32	0.36	2.74
wupwise	OMP2001	F	2468	43	1.37	5.68
libquantum	Spec2006	C	2743	73	1.30	6.30
leslie3d	Spec2006	F	3024	32	8.24	23.72
namd	Spec2006	C++	4077	193	19.46	111.53
astar	Spec2006	C++	4377	111	1.49	8.39
bzip2	Spec2006	C	5896	51	4.8	90.27
milc	Spec2006	C	9784	172	41.16	19.68
sjeng	Spec2006	C	10628	121	9.93	34.98
sphinx	Spec2006	C	13683	210	7.11	31.19
zeusmp	Spec2006	F	19068	68	37.85	285.48
omnetpp	Spec2006	C++	20393	3980	21.66	58.24
hmmmer	Spec2006	C	20973	242	12.13	36.52
soplex	Spec2006	C++	28592	1523	21.21	144.14
h264	Spec2006	C	36495	462	29.56	220.53
cactus	Spec2006	C	60452	962	25.65	185.05
gromacs	Spec2006	C/F	65182	674	47.82	252.33
dealII	Spec2006	C++	96382	15619	114.30	240.18
calculix	Spec2006	C/F	105683	771	192.99	404.32
povray	Spec2006	C++	108339	3678	71.01	242.61
perlbench	Spec2006	C	126367	2183	94.18	210.37
gobmk	Spec2006	C	157883	4188	60.66	242.19
gcc	Spec2006	C	236269	6426	280.37	490.68
xalan	Spec2006	C++	267318	30,062	264.97	183.75
gzip	Compress	C	10671	98	1.42	20.06
tar	Compress	C	20518	343	9.58	18.85
ssh	Web clinet	C	73335	887	40.57	22.55
lynx	Browser	C	135876	2106	140.08	73.01
apache	WebServer	C	232931	2026	37.98	232.12

Table 4.2: *Applications Table*

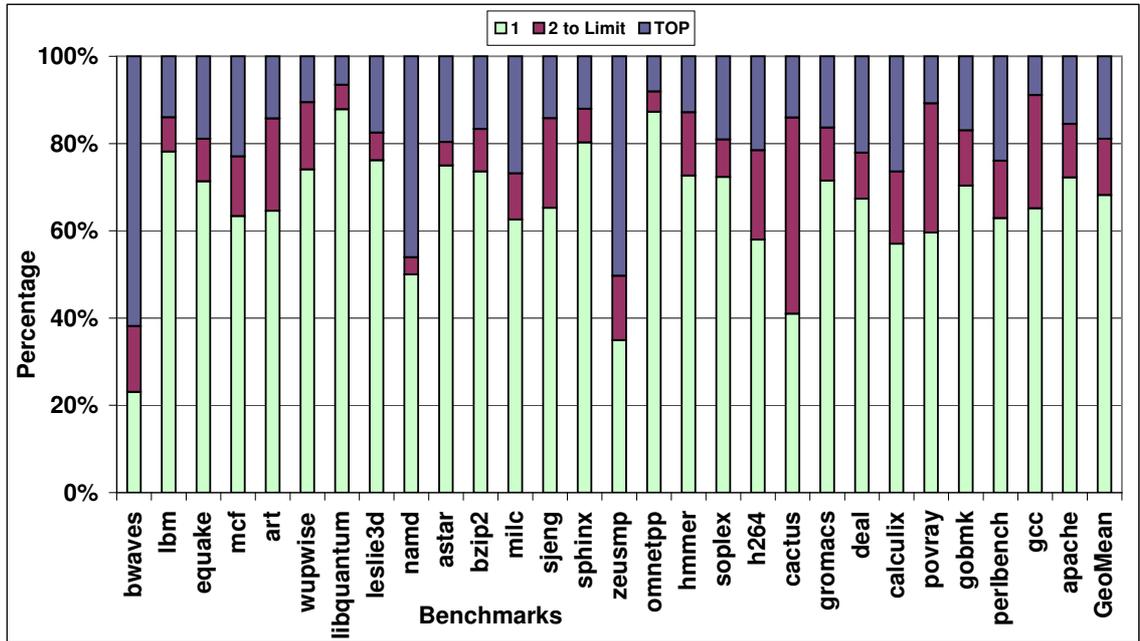


Figure 4.10: *Symbolic Value Set Visualization*

Fig 4.10 presents an insightful result regarding the functioning of the symbolic analysis. It divides the objects into various categories according to the size of their symbolic value set in our abstract domain. On average, around 64% of objects can be abstracted with a single symbolic expression in our symbolic domain, 16% of objects need multiple expressions and 20% of objects cannot be represented with finite symbolic abstraction (\top , referred as *TOP*). Maintaining a symbolic value set instead of a single symbolic expression allows us to maintain this extra precision for 16% of data objects.

In order to understand the importance of tracking memory locations, we obtain the fraction of symbolic expressions that containing at least one symbolic alphabet propagated through a memory location, out of symbolic expressions for all IR variables. Fig 4.11 shows that 35% of symbolic expressions contain alphabets propagated through memory locations. In absence of an abstraction for memory locations, the

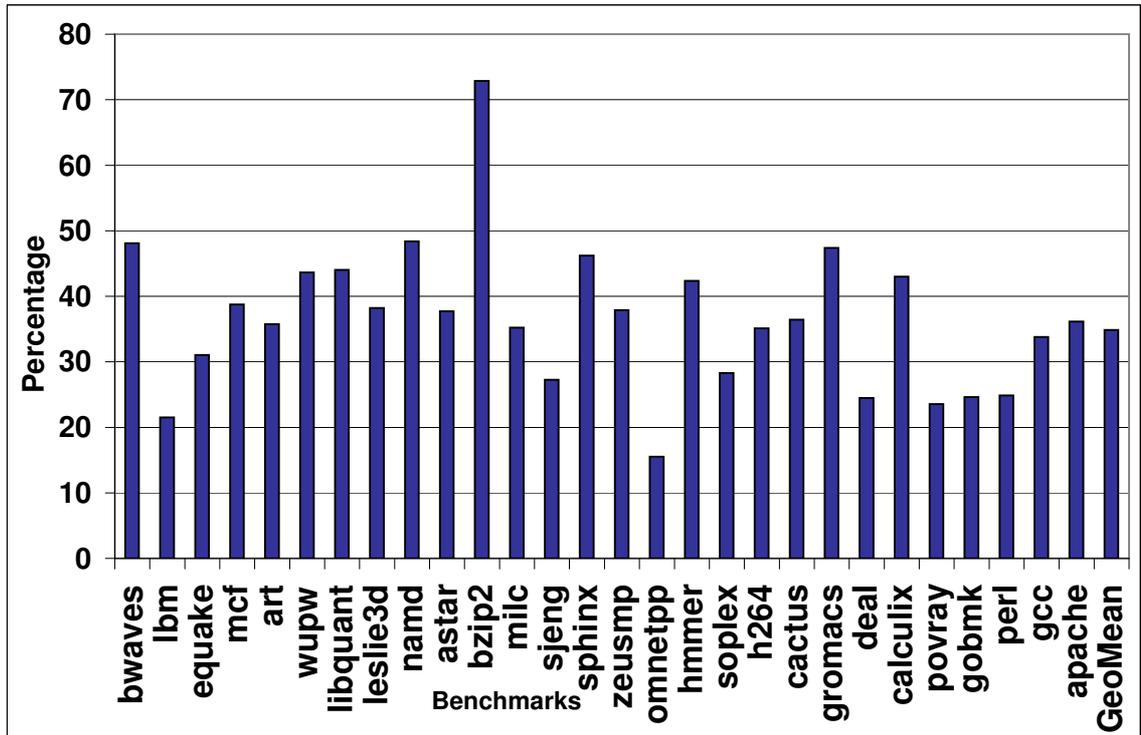


Figure 4.11: *Percentage of symbolic expressions that containing at least one symbolic alphabet propagated through a memory location, out of symbolic expressions for all IR variables*

analysis would have introduced a new alphabet in all these expressions according to the rules in Table 4.1. This validates our central contribution that tracking memory locations is essential for effective symbolic analysis on executables.

Fig 4.12 highlights another interesting aspect of our symbolic analysis framework. It presents the percentage of data objects which are represented as $TOP(\top)$ in the symbolic abstraction for different choices of the maximum size of a symbolic value set ($LIMIT$). This result shows that percentage of unrepresentable data objects reach a stable point and does not decrease further with increase in $LIMIT$, validating our choice of $LIMIT(5)$ for representing the symbolic abstractions.

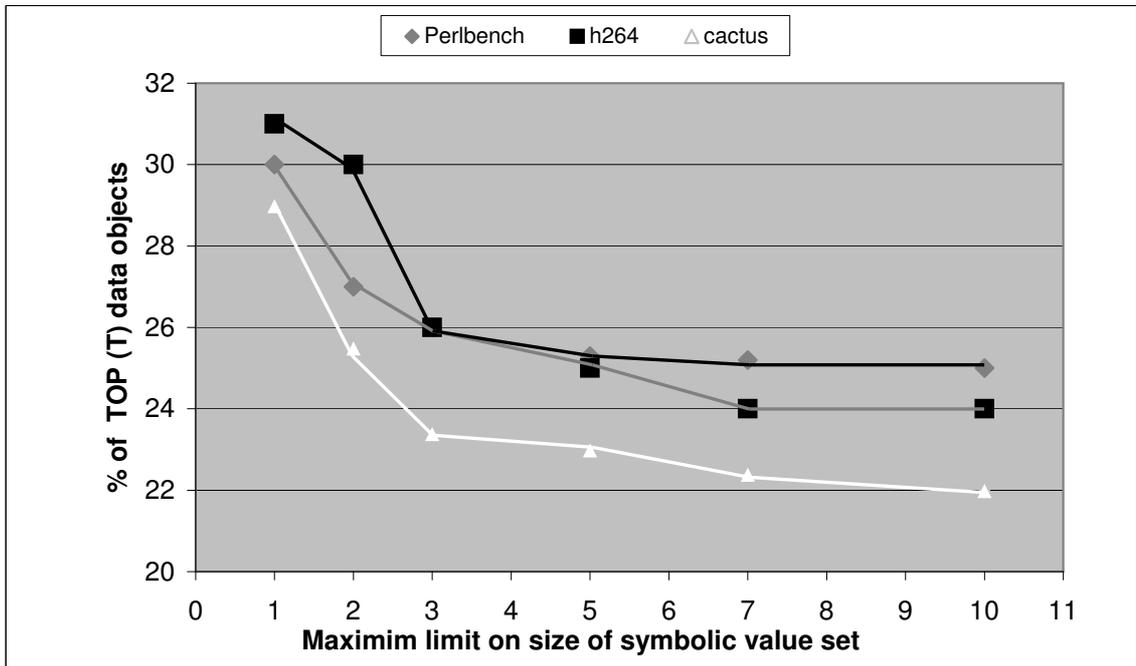


Figure 4.12: *Variation of TOP (τ) data objects with varying size of symbolic value set*

4.9.2 Value numbering

We implemented the value numbering algorithm as presented in Section 4.8 for determining equivalent computations and for eliminating redundant computations from the executables. Fig 4.13 compares the number of equivalent computations determined in three cases: one when no symbolic analysis is performed, second when symbolic analysis is employed only for variables (obtained by neglecting the transfer functions for memory load and memory store in Table 4.1) and third, when memory based symbolic analysis is employed to determine equivalence. Hence, the second case is similar to existing source-level methods of symbolic analysis since it tracks only variables. The third case represents our contribution since it tracks memory locations as well. As evident from this figure, value numbering employing memory-

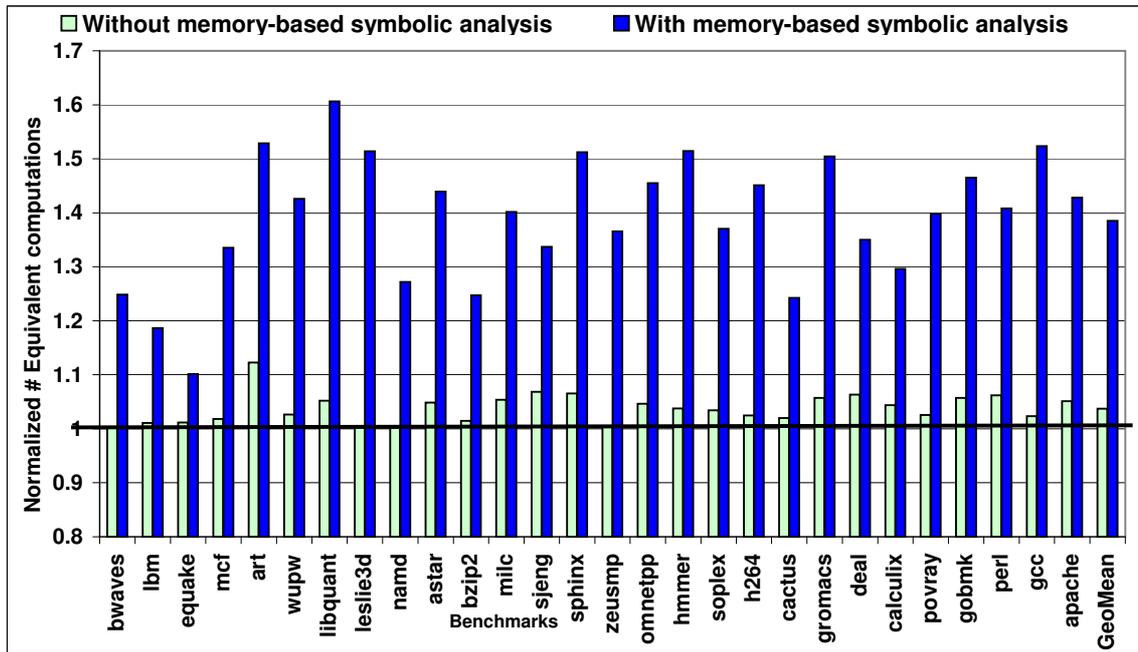


Figure 4.13: *Normalized improvement in detection of equivalent computations (No Symbolic analysis = 1.0)*

based symbolic analysis is able to expose around 40% more equivalent computations in executables than base value numbering (when no symbolic analysis is applied). This figure also shows that symbolic analysis based only on variables is not sufficient in exposing more equivalences in executables and exposes only 3% more equivalences than discoverable when no symbolic analysis is applied. This validates our central contribution – that tracking memory locations is essential to get good results for symbolic analysis on executables.

Fig 4.14 compares the static counts of redundant instructions removed in the three scenarios above and shows that memory-based symbolic analysis improves the removal of redundant computations by around 32%. The removal of redundant computations will improve the efficiency and efficacy of any subsequent binary analysis and will also improve the runtime of rewritten binary in case this analysis in

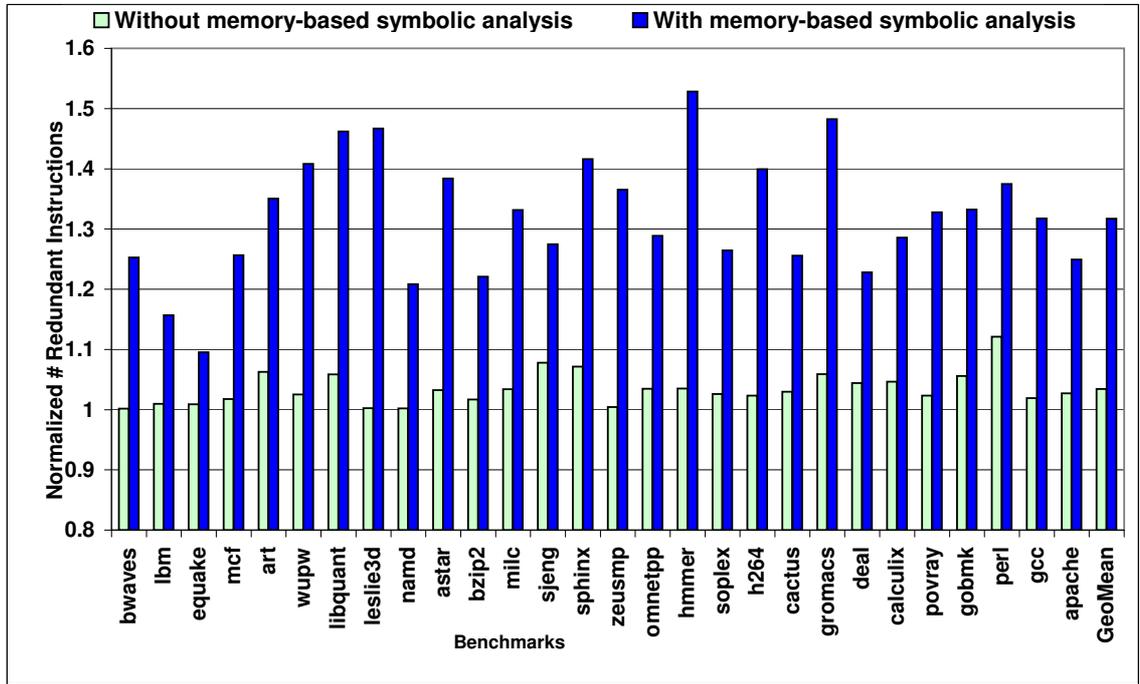


Figure 4.14: *Normalized improvement in removal of redundant instruction (No symbolic analysis=1.0)*

employed in a binary rewriter.

4.9.3 Program parallelization

Next, we substantiate the impact of symbolic analysis on dependence tests for the purpose of program parallelization. As presented in Section 4.7, various parallelization methods first try to disprove dependence between pairs of subscripted references to the same array variable and next try to characterize the dependence by determining the distance vector between two accesses to the same memory location in terms of number of loop iterations. The dependence tests are considered to be successful when a precise answer can be obtained for any of the above tests.

We implemented various common dependency tests like ZIV tests (Zero induc-

Application	Suite	# Tests	#Success (Without mem based sym analysis)	#Success (With mem based sym analysis)	% Imp
2mm	Polybench	18	14	18	28.5
3mm	Polybench	26	20	26	30.0
atax	Polybench	6	3	4	33.3
bicg	Polybench	13	6	10	66.7
covariance	Polybench	19	16	19	18.7
doitgen	Polybench	25	10	23	130.0
gemm	Polybench	14	12	14	16.67
gemver	Polybench	26	22	26	18.18
gesummv	Polybench	13	9	12	33.3
jacobi	Polybench	27	13	13	0
ft	NAS	127	37	43	16.2
lu	NAS	4866	1438	2078	44.5
bt	NAS	2866	1844	2237	21.3
sp	NAS	3317	2287	2815	23.1
AVG					34.3

Table 4.3: *Parallelization benchmarks*

tion variable), SIV test (Single induction variable), and MIV test (Multiple induction variables) as presented in [69]. We measured the number of array references where any of the dependence tests was able to eliminate dependence, or was able to provide a precise answer to the distance between dependencies.

We have tested our framework on executables of benchmarks from the Polyhedral Benchmark suite [115] and the NAS benchmark suite [108]. Table 4.3 describes the usage and success frequency of dependence tests for each of the benchmarks. It lists the number of times the test was applied in each benchmark and the number of times the test was able to give a precise answer in two situations: using the memory based symbolic analysis and using only variable based symbolic analysis. Since dependence tests rely on affine expressions for loop indices, none of the dependence tests succeed when no symbolic analysis is applied. Hence, we omit the results for the case of no symbolic analysis. This table shows that the memory based symbolic analysis framework improves the precision of standard dependence tests on execu-

bles by 34% on average. The improvement in the precision of dependence tests will further enhance the ability of binary-level parallelizers.

Existing binary-based parallelization techniques [93, 162] implement custom methods to recognize induction variables from binaries. These techniques are orthogonal to our method, consequently, we have not compared our techniques with these methods. Nonetheless, our symbolic analysis framework can obviate the need for any custom induction variable recognition method for binary-parallelization.

4.9.4 Alias analysis

As mentioned in Section 4.3.3, although Value Set Analysis (VSA) is a powerful alias analysis algorithm for executables, there are a few scenarios where the symbolic abstraction can aid VSA abstraction in resolving aliasing queries. We identified a few scenarios where VSA yields imprecise answer to aliasing queries. One example of such a scenario is allocation of arrays with statically unknown size since VSA does not declare memory abstractions for such arrays. The underlying reason is that their algorithm relies on an Aggregate Structure Identification algorithm [120] which requires constant static bounds of arrays to represent array access constraints for constraint solvers. Another common example is accessing an element of an array with an input dependant index.

In order to evaluate the effectiveness of our symbolic analysis, we compare it with VSA in only those portions of binary code where VSA results in imprecise answers due to the above limitations. In order to identify such locations in the

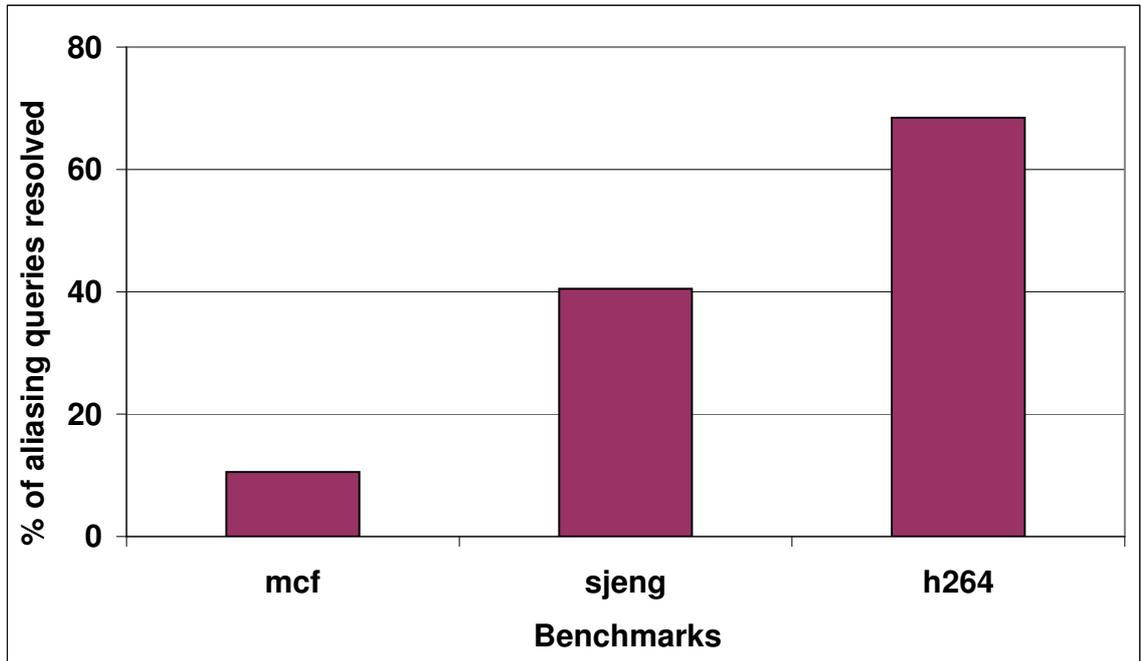


Figure 4.15: *Alias analysis results*

code, we implemented a custom data flow analysis that identifies whether the memory address accessed in a memory access instruction is input dependent (statically unknown, hence \top in VSA). The analysis begins by defining the variables resulting from known I/O external function (e.g. `fread`) as input dependant. This information is then propagated interprocedurally across the whole program in a method similar to program slicing method presented by Horwitz et. al. [83]. The above data-flow analysis is implemented on the source code to identify the code locations where VSA yields imprecise answers to aliasing queries. The symbolic analysis and VSA results are compared in the corresponding code locations in the executables.

Fig 4.15 lists a few sample benchmarks for our custom data flow analysis identified a significant number of code locations. For each benchmark, Fig 4.15 presents the percentage of aliasing queries which symbolic analysis was able to re-

solve (must-alias or no-alias); among queries for which VSA provided an imprecise answer (may-alias). As evident from the figure, our method can resolve between 10% and 65% of aliasing queries that could not be resolved using VSA alone. This result establishes that symbolic analysis can aid VSA in improving the precision of aliasing decisions in executables. In the above examples, we found no improvement in the resolution of aliasing queries when we switched off the transfer functions for memory loads and stores – this shows the importance of tracking memory locations for improved alias analysis as well.

Chapter 5: **Improving memory abstraction**

5.1 Precise Memory Model

Executable specific artifacts such as indirect call transfer instructions (CTI) complicate the task of recovering a *precise memory abstraction* while maintaining the *functionality* in IR. A memory abstraction involves associating each stack memory reference to a set of variables on the memory stack. This is useful since most program analysis techniques require variables rather than memory locations. In order to recover such an abstraction, we need to determine the value of stack pointer at each program point in a procedure relative to its value at the entry point. This is usually accomplished by analyzing each stack modification instruction, including CTIs which can possibly modify the stack pointer due to several reasons such as cleanup of arguments passed on the stack.

However, the modification in the value of stack pointer cannot be easily determined in all scenarios. For example, in case of an indirect CTI, the stack modification is deterministic only if all its statically determined possible targets modify the stack pointer by the same value. However, such targets might modify the stack pointer by different values, or a call to an external function with an unknown pro-

prototype might have a statically indeterminable impact on the value of stack pointer. Existing frameworks such as IDAPro [84] and CodeSurfer/X86 [21] require that the return from a CTI should always modify the stack pointer by a deterministic constant value. In above mentioned scenarios, CodeSurfer/X86 recovers an imprecise memory abstraction that does not associate stack memory references to variables on stack, hurting the analyzability of IR. In contrast, IDAPro aims to recover a precise memory abstraction; but when it cannot, it makes unsafe assumptions yielding a non-functional IR.

We present techniques for recovering a precise memory model and functional IR in such scenarios. Our mechanism formulates a set of constraints using control flow constructs in the caller procedure to compute the value of stack modification at a call-site. The constraints are solvable in most scenarios. When the constraints cannot be solved, it embeds run-time checks to maintain the functionality of IR. This enhanced memory model improves the precision of several analysis techniques for executables.

5.2 Motivation

In this section, we demonstrate the limitation of existing frameworks in obtaining a functional IR with a precise memory model and the relative importance of considering the underlying memory model for symbolic abstraction.

A source program has an abstract stack representation where the local variables are assumed to be present on the stack but their precise layout is not specified.

In contrast, an executable has a fixed physical stack layout.

To recreate an IR, the physical stack must be deconstructed to individual abstract frames, one per procedure. Since, each frame comprises variables from the source code, a memory model is defined as precise if each such frame can be divided into abstract locations analogous to the original variables.

Previous methods [21] have approached this problem in two steps. First, all the instructions in a procedure which can modify the stack pointer are analyzed to compute the maximum size to which the stack can grow in a single invocation of the procedure among all its control-flow paths. Next, each such abstract frame is further abstracted through a set of `a-locs`. An `a-loc` is characterized by two attributes: its relative offset in the region with respect to other `a-locs` and its size. The `a-loc` representation requires the determination of the value of the stack pointer at each program point in a procedure relative to its value at the entry point.

As highlighted in Section 5.1, this is usually accomplished by tracking each update to the stack pointer. However, several artifacts might result in a non-deterministic stack modification, invalidating the inherent assumption in previous frameworks [21]. Next, we analyze such scenarios in more detail. We characterize the impact of a CTI `I` on the value of stack pointer by introducing the following definition:

$\text{StackDiff}(I) = \text{Stack Pointer after } I - \text{Stack Pointer before } I.$

The term `StackDiff` can be applied to either the CTI or a corresponding called procedure, and represents the stack modification amount in either case. `StackDiff` of a CTI can be positive if the called procedure cleans up its arguments, or zero

if it does not. In theory, it can be negative if the procedure leaves some local allocations on the stack, although we have not observed this in compiled code. Several approaches have been suggested to calculate the value of `StackDiff` by symbolically evaluating all the stack modification instructions in a procedure [21]. As per these methods, `StackDiff` at an indirect CTI is deterministic if all possible targets have the same value of `StackDiff`. Thereafter, the stack pointer in the caller procedure is adjusted by `StackDiff` amount. This adjustment is imperative for maintaining the correctness of data-flow in caller procedure.

However, `StackDiff` cannot be determined statically in all scenarios. For example, possible targets of an indirect CTI might have different `StackDiff`, or an external function with an unknown prototype might have a statically unknown `StackDiff`. In such scenarios, existing frameworks either result in an imprecise memory abstraction or fail to maintain the correctness. As per CodeSurfer/X86, “if it cannot determine that the change is a constant, it issues an error report” (Section 4.2) [21]. Hence, the corresponding frame cannot be represented through `a-locs`, resulting in an imprecise memory model. IDAPro applies a constraint-based mechanism to compute the values of `StackDiff` independent of the called procedures. However, when the underlying method fails to determine a unique solution, it compromises the correctness by accepting one feasible solution (which could be wrong) out of an infinite number of possible outcomes [133].

Fig 5.1 illustrates an example of such a scenario. In Fig 5.1, a local region of size 24 is allocated in a procedure, consequently, the memory access at Line 2 results in the discovery of an `a-loc` at offset 16. Suppose the possible targets of the

```

main:
1  sub 24, $esp           //Local Allocation
2  mov $10, 8(%esp)     //Access (%esp+8)
3  call *%eax           // An Indirect call
4  mov $20, 12(%esp)    //Access
                          //( %esp+12+UNKNOWN)
      .....

```

Figure 5.1: *An example demonstrating the imprecision in the presence of indirect calls, second operand in the instruction is the destination*

indirect CTI at line 3 have different `StackDiff` values. Consequently, `esp` after Line 3 has an unknown offset relative to its value at the entry point of the procedure. Hence, no `a-loc` can be identified at Line 4. On the other hand, if `StackDiff` value is calculated wrongly, it results in an incorrect data-flow at Line 4.

Our hybrid mechanism maintains the precision as well as functionality. Our static mechanism enables abstraction through a set of `a-locs` and dynamic mechanism guarantees the correctness when `StackDiff` cannot be computed.

5.3 Recovering precise memory abstraction

In this section, we discuss our hybrid static-dynamic solution to obtain a functional representation with a precise memory model. We first present a symbolic constraint mechanism to determine the value of `StackDiff` for each CTI where it is unknown. Next, we discuss our solution for maintaining the functionality even when `StackDiff` at some CTIs cannot be solved. Our analysis employs the prototypes of well-known library functions, similar to to the IDAPro’s FLIRT database [84], for determining

their `StackDiff` value. We assume that existing methods [21] are able to determine the value of `StackDiff` for each procedure, which holds true under the assumptions of *standard compilation model*.

5.3.1 Static Computation

A CTI I can result in an unknown `StackDiff` in three cases, which we collectively refer to as *Unknown CTIs*.

Case 1: I is a direct CTI to an external procedure with unknown prototype.

Case 2: I is an indirect CTI with unresolved targets.

Case 3: I is an indirect CTI and its targets have different `StackDiff`.

In such scenarios, our symbolic constraints based mechanism employs several boundary conditions imposed by the control flow inside the corresponding caller procedure to determine `StackDiff`. The proposed constraint formulation does not require us to determine the precise set of targets of an indirect CTI, which itself is an extremely challenging problem.

We define symbolic values X_I and S_I for representing `StackDiff` and local stack height at a CTI I . Every stack modification instruction in a procedure is analyzed to derive an expression of S_I in terms of the X_I s. The resulting expressions are transformed into a linear system of equations that can be solved to calculate the value of X_I s.

Fig 5.2 presents the rules for generating symbolic constraints and equations in a particular procedure P . It presents rules for analyzing each stack modification

Unknown Symbolic Values : X_I , where $X_I = \text{StackDiff}$ of procedure call I

Initial/Helper Variables :

$\text{Targ}(T)$: Set of procedures targeted by call target address T

$\text{StackDiff}(f)$: StackDiff of procedure f

$Y_SET(F) = \cup_{f \in F} \text{StackDiff}(f)$

BeginP = Entry point of procedure P; Pred_{BB} = Predecessors of basic block BB;

$\text{BeginBB}, \text{EndBB}$ = Entry point, terminator of basic block BB

S_I = Stack height after instruction I;

S_{BB} = Stack height at beginning of basic block BB;

PrevI = the previous instruction to I ($I \neq \text{BeginBB}$)

$S_{I'}$ = if ($I \neq \text{BeginBB}$) then S_{PrevI} else S_{BB}

R : A register, $\text{Size}(R)$: Size of register R, N: A constant

Initial Conditions : $S_{\text{BeginP}} = 0$

Data flow rules :

For every instruction I:

I = push R $\Rightarrow S_I = S_{I'} + \text{size}(R)$

I = pop R $\Rightarrow S_I = S_{I'} - \text{size}(R)$

I = add esp, N $\Rightarrow S_I = S_{I'} - N$

I = sub esp, N $\Rightarrow S_I = S_{I'} + N$

I = jmp L $\Rightarrow S_{\text{BeginL}} = S_{I'}$

I = call Y \Rightarrow

if ($Y_SET(\text{Targ}(Y))$ contains a single constant C)

$S_I = S_{I'} + C$

else

$S_I = S_{I'} + X_I$

default (if not an invalidation condition) $\Rightarrow S_I = S_{I'}$

Boundary Conditions :

1. $\forall BB: \forall \text{Pred} \in \text{Pred}_{BB}, S_{\text{BeginBB}} = S_{\text{EndPred}}$

2. I = ret : Constraint $S_{I'} = 0$

Invalidation Conditions :

1. I = esp \leftarrow ... /* Any assignment except in data-flow rules*/

2. I accesses return address

Figure 5.2: Data flow rules used to determine stack modifications in a procedure P

instruction, a set of initialization and boundary conditions for solving the symbolic equations and a set of conditions which invalidate our symbolic constraints for the current procedure.

In an x86 program, several instructions can modify the value of stack pointer. The local frame in a procedure is usually allocated by subtracting a constant value from `esp`. Similarly, the local frame is deallocated by adding a constant amount to `esp`. Push and pop instructions implicitly modify the stack pointer by the size of amount pushed onto the stack. The rules in Fig 5.2 incorporate the deterministic modification at each CTI. An indeterministic modification is modeled symbolically as X_I . The dataflow rules in Fig 5.2 obtain an expression for S_I considering each such stack modification instruction.

In order to solve the symbolic equations obtained through dataflow rules, Fig 5.2 generates two constraints based on the control flow in procedure P. These conditions hold true for every executable following the standard compilation model [21]:

- $\forall \text{Pred} \in \text{Pred}_{\text{BB}}, S_{\text{BeginBB}} = S_{\text{EndPred}}$: This condition implies that at a merge point in the control flow of a procedure, the stack height at the end of every predecessor basic block must be equal. Otherwise, any subsequent stack access might access different stack locations depending on the path taken at run time, resulting in an indeterminate behavior.
- $S_I = 0 \forall \text{ret} \in P$: In an x86 program, a return instruction loads an address from the location pointed by `esp` and sets the program counter to the loaded value. Since the return address is pushed by the caller procedure and

a compiled program usually does not access the return address directly, `esp` can refer to the return address only if stack height S_I is zero. Thereafter the return instruction may optionally specify an operand to clean up some incoming arguments, so `StackDiff` could be positive or zero.

Fig 5.2 also formulates the following conditions which invalidate the assumptions behind our boundary conditions. In such situations, we discontinue our static mechanism and rely on our dynamic mechanism to maintain the correctness of IR.

- `I = esp ← ...` : Any assignment to `esp` other than those in data-flow rules implies a local frame allocation of variable size. In such a scenario, the boundary conditions fail to obtain a solution for X_I . However, this condition arises in extremely rare circumstances of variable size arrays on stack frame.¹
- `I` accesses return address: In a usual compiled code, `StackDiff` is either zero or positive. In theory, procedures could have a negative `StackDiff`, implying that the procedure leaves some local allocations on the stack. In such scenarios, `esp` would not point to the return address at the point of return. Hence, a return must be implemented by explicitly accessing the return address from the middle of the stack. This invalidates the assumption behind our boundary condition 2 and we resort to run-time checks.

The resulting symbolic equations are solved by employing a custom linear solver that categorizes the equations into disjoint groups based on the variables

¹Code produced by popular compilers contains x86 idioms like `leave` instruction which implicitly assign a previously stored value to `esp`. Such idioms are currently handled explicitly in our framework.

used in every equation. A group is solved only if the number of equations is equal to the number of unknowns. We keep propagating calculated values to other groups until no more calculated values are present. Once we obtain a solution of X_I for each I in a procedure, we can obtain a safe abstraction of abstract memory regions into a set of *a-locs* using the methods in [21, 74].

5.3.2 Dynamic Mechanism

As mentioned above, the above method does not guarantee a solution for all the scenarios. For example, the above method fails to determine the value of `StackDiff` in basic blocks containing multiple CTIs each with an unknown X_I value. Below, we discuss our dynamic mechanism to handle all the three cases of *Unknown CTIs* presented in Section 5.3.1.

Case 1: Since this case represents control transfer to an external procedure, the body of the called procedure cannot be modified. Such scenarios are handled by employing a trampoline mechanism to call the external procedure. The trampoline dynamically computes the shift in stack pointer value before and after the call using inline assembly instructions.

Case 2 and Case 3: Recall from Section 4.4 that an indirect CTI is translated to the corresponding location in IR using a switch statement inside a *call translator* procedure. In such scenarios, `StackDiff` is declared as an explicit return variable in the prototype of call translator procedure. The definition of the call translator is modified to return the value of `StackDiff` for the called procedure in each switch

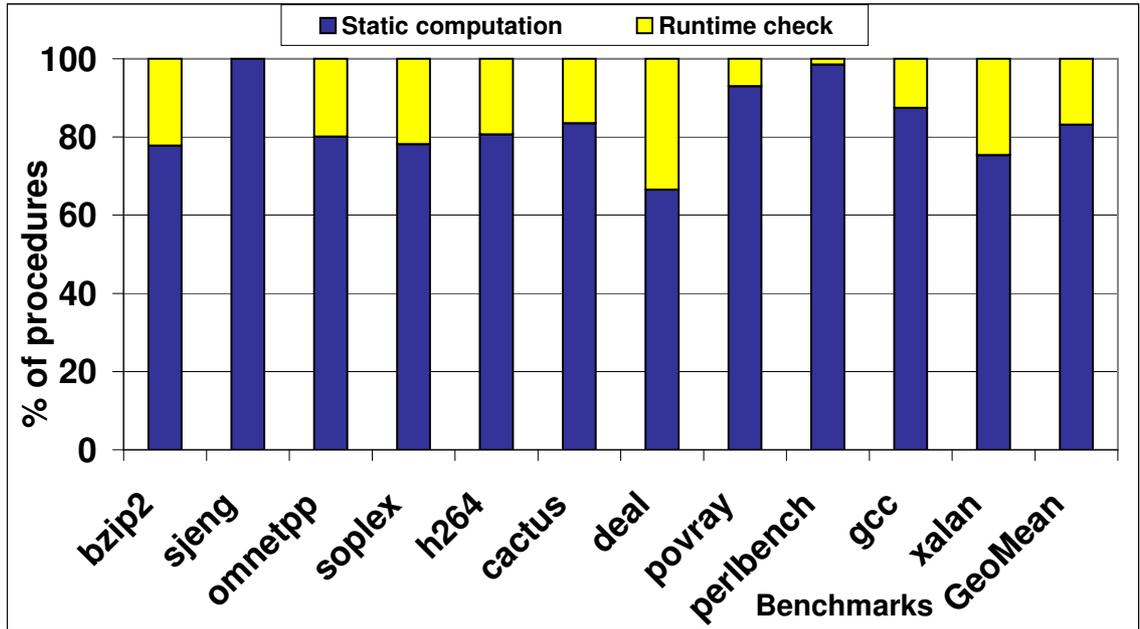


Figure 5.3: Percentage of procedures with unknown CTIs. The static represents cases when constraint solvers succeed

statement.

5.4 Results

Fig 5.3 and Fig 5.4 present the statistics regarding our hybrid mechanism for obtaining precise memory model and functional IR. We only present statistics for benchmarks containing non-negligible *Unknown CTIs* (negligible defined as ≤ 10 or number of procedures containing *Unknown CTI* $\leq 1\%$). Of 33 programs in Table 4.2, 11 had non-negligible unknown CTIs. Fig 5.3 presents the fraction of procedures containing *Unknown CTI* in each of these benchmarks. It divides this fraction into scenarios where the static mechanism was able to determine the value of `StackDiff` and where the dynamic mechanism was required to maintain the functionality. Case 1 (Section 5.3.1) does not arise since we employ the prototypes for standard library

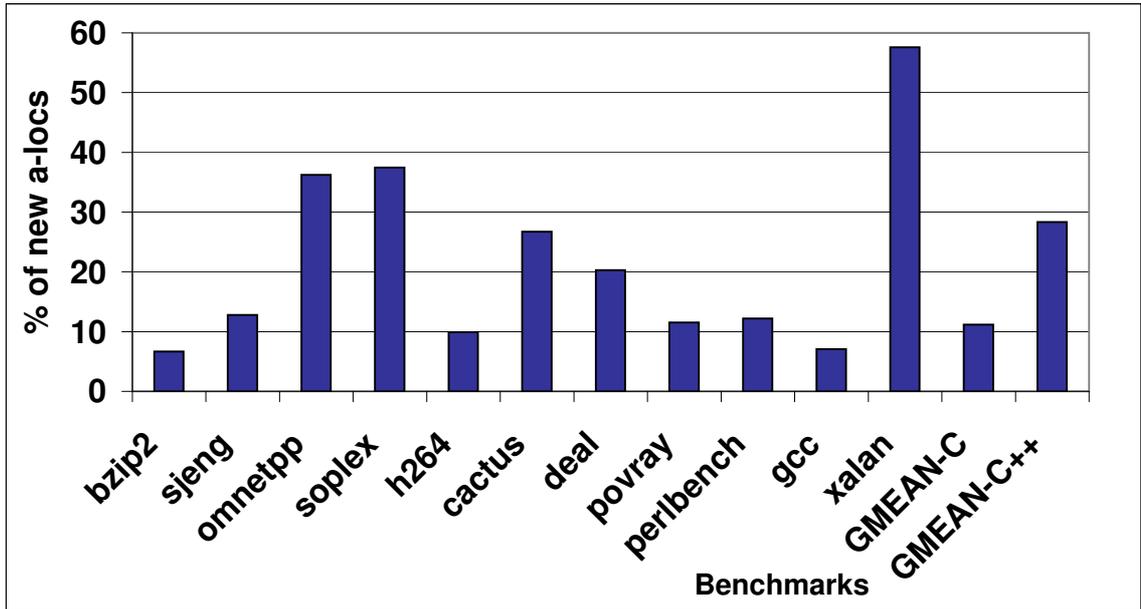


Figure 5.4: *Additional alocs added as a result of constraint solvers, normalized to original number of alocs*

procedures. We never hit the invalidation conditions stipulated in Fig 5.2, further justifying the assumptions behind our constraint formulation.

Fig 5.4 illustrates the additional a-locs derived as a result of successful constraint solutions, normalized with respect to original a-locs of type `Stack`. As evident, we were able to obtain 10% more `a-locs` in C benchmarks and 30% more `a-locs` in C++ benchmarks on average. This result reinforces the relative importance of our mechanism in C++ benchmarks. This enhanced `a-locs` abstraction is employed in our symbolic value analysis framework.

Fig 5.5 captures the enhancement in the precision of Symbolic Value Analysis with the presence of additional `a-locs` derived by the constraint mechanism. According to the rules in Table 4.1, a load instruction accessing an unknown memory location is represented by a new symbolic alphabet. Fig 5.5 demonstrates the

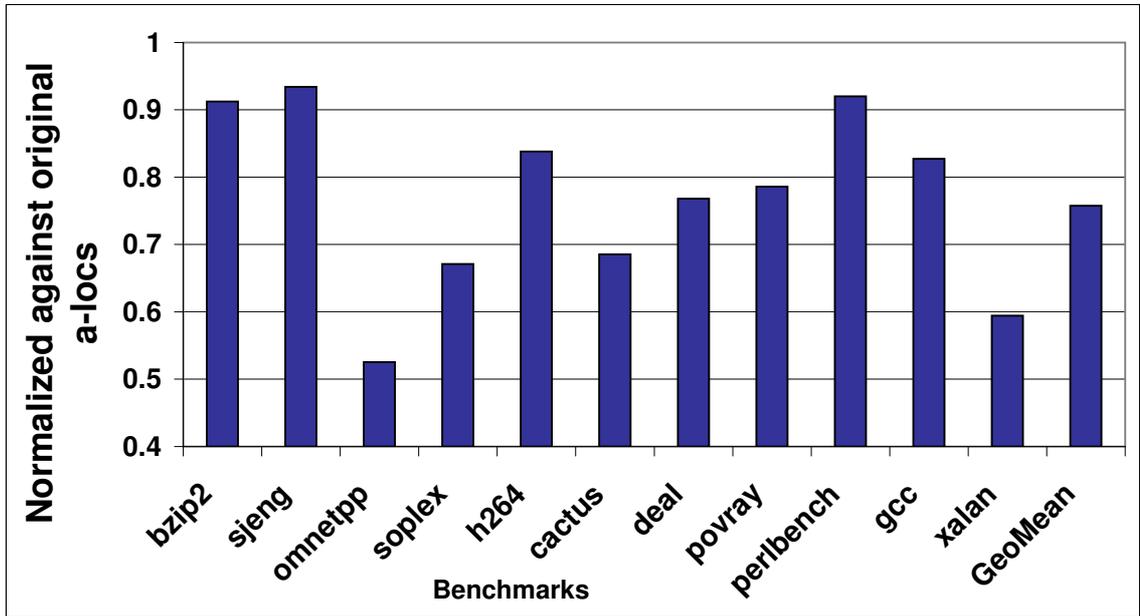


Figure 5.5: Variables requiring a new symbolic alphabet in presence of additional *a-locs*

decrease in the number of load instructions requiring a new alphabet while employing additional *a-locs*. The presence of additional *a-locs* enhances the precision of symbolic value analysis by 10% to 50% in several programs.

Chapter 6: **Information flow security of executables**

6.1 Introduction

The rapid rise in cyberattacks has exposed serious security vulnerabilities in software systems. Information flow violations collectively comprise one of the most critical vulnerabilities in this regard. Such violations subject the programs to severe security attacks like format-string attacks [132], directory-traversal attacks [56], cross-site scripting, SQL-injection [143] and also result in the leakage of confidential and sensitive information to untrusted parties [147].

Research has led to a number of approaches proposed to mitigate the susceptibility arising due to information flow violations in programs. The most popular of these methods model the violations as violations to an information flow policy, and enforce the policy through a tracking mechanism [110, 119, 64, 50, 87, 163]. The inherent idea is to mark the untrusted (or confidential) information in a program as *tainted*, propagate the tainted labels through the program's data and control flow and enforce the required policy by raising an alarm at every illegal use of the tainted information.

Despite significant research efforts, existing approaches fall short with regards

to several desired characteristics of security techniques: *practicality*, defined as the ability to handle off-the shelf programs without any performance overhead; *precision*, defined as the ability to uncover policy violations without excessive false positives; *scalability*, defined as the applicability to large real-world programs; and *extensibility*, defined as the ease of handling multiple security policies and the ability to counter rapidly-evolving threats.

Regarding *practicality*, several existing information flow tracking systems are constrained by their underlying frameworks. A significant number of previous information flow systems either leverage dynamic binary frameworks or statically analyze the applications written in high-level languages like C, C++. Both the approaches have limited applications in real-world scenarios, as discussed next.

Information-flow tracking based on dynamic frameworks [110, 119, 64, 50] experiences high runtime overhead unless the tracking mechanism is implemented in hardware [56]. Further, dynamic frameworks only detect the violations arising in a single execution path. Therefore, static techniques seem to present an appealing alternative.

However, most static information flow systems detect violations from programs written in high-level languages [143, 132]. These methods have limited applicability in many real-world scenarios where the source-code of third-party and proprietary executables is not available to end-users who want to protect their systems. Further, it is a well known fact that compilers are a source of vulnerabilities in programs [21] and source-code analysis is insufficient in detecting the violations arising due to compiler-introduced bugs.

There has only been limited work in uncovering vulnerabilities in executables using static mechanisms, and these have their own drawbacks. The existing static mechanisms for detecting vulnerabilities in executables fail to combine *precision* and *scalability*. Such frameworks [53, 155, 61] ignore memory and aliasing issues, resulting in an imprecise analysis and limited vulnerability detection. As demonstrated by existing source code mechanisms [101], a precise points-to analysis is imperative for achieving a low false positive rate. Analyzing memory accesses is even more essential for executables than source code since executables mainly contain memory locations instead of explicit program variables. The paucity of registers in x86 ISA further underscores this requirement. Hence, ignoring aliasing issues limits the capability of existing tools for executables to reliably expose vulnerabilities without plaguing the results with false alarms.

However, analyzing memory accesses in an elementary manner in executables might adversely impact the *scalability* of the system. As observed in several source-level frameworks, an exhaustive analysis of memory accesses constrains the *scalability* of the underlying system [79, 143]. Hence, previous source-level information flow systems [143, 53, 132] balance precision and scalability in the presence of pointer operations by employing innovative frameworks such as thin slicing [143] and type inference mechanisms [132]. However, no counterpart methods of such frameworks have been proposed for executables, resulting in severe precision and scalability challenges for static executable analysis systems.

Further, several existing information flow frameworks lack *extensibility* since the underlying single-bit taint tracking mechanism cannot be extended to detect var-

ious advanced information flow violations. Even though single bit taint mechanisms are more efficient in terms of memory usage, they lack the ability to enforce multiple policies concurrently, which limits its capability to protect against the attacks that exploit multiple vulnerabilities in an orchestrated manner [139]. As presented by Chang et al. [43], single-bit frameworks cannot expose file-disclosure vulnerabilities.

We present DemandFlow, a novel information flow mechanism for executables to address the above limitations of *practicality*, *precision*, *scalability* and *extensibility*. DemandFlow eliminates a major limitation of existing static information flow frameworks for executables by employing *precise* mechanisms for propagating information across memory locations. In addition, DemandFlow boosts *scalability* by propagating information across only those program variables and memory locations which are critical for the flow of information regarding a particular policy. Instead of analyzing the whole program, DemandFlow proposes a novel demand-driven analysis tailored to an actual policy.

Several demand-driven mechanisms have been proposed for popular compiler analyses like pointer analysis and interprocedural data flow analysis [79, 59]. One of our major contributions is the application of such popular compiler scalability concepts to address the precision and scalability challenge in information flow analysis of executables.

DemandFlow also provides an easily *extensible* mechanism for detecting several kind of vulnerabilities. We note that information flow tracking is a special case of program data-flow analysis. Hence, instead of propagating a single-bit taint information, DemandFlow computes an information abstraction which can be easily

extended to represent several different policies. The analysis cost for maintaining an information abstraction, instead of a single-bit taint information, is ameliorated by the ensuing simplification while enforcing multiple policies simultaneously in a program. Further, our information abstraction enables the attribution of violation of a policy to the culprit information source.

DemandFlow is used as an analysis tool, similar to how static analysis mechanisms are used, complementing other bug-finding tools and dynamic information flow trackers. The primary contributions of our work are the following:

- **Precise Static Analysis:** DemandFlow employs a powerful static analysis mechanism that precisely handles memory aliasing issues in executables. Previous static information flow systems for executables ignore aliasing issues resulting in an imprecise analysis.
- **Demand-driven Framework:** DemandFlow achieves scalability while reasoning about memory accesses by tailoring the analysis to a particular information flow policy. Instead of doing an exhaustive analysis over the complete program, it computes the set of program objects critical for preserving the flow of information with respect to a particular information flow policy and propagates information for only such program objects.
- **Diverse Evaluation:** We apply DemandFlow on several information flow violations such as format string attacks, directory traversal attacks, and information flow leakage. DemandFlow uncovers six previously undiscovered format string and directory traversal vulnerabilities in popular `ftp` and inter-

net relay chat programs. It also exposes an unknown information (password) leakage vulnerability on KeePassX, a popular password manager application. DemandFlow reliably detects previously known vulnerabilities in a variety of real-world programs at a low false positive rate of approximately 1 per 20,000 lines of code. DemandFlow is scalable and analyzes large programs such as *MySQL* (1.7 million lines of code) in around 7 minutes. The scalability is further demonstrated by evaluating DemandFlow on all the programs in SPEC2006 suite.

The rest of the chapter is organized as follows. Section 6.2 discusses the related research work. Section 6.3 presents an overview of DemandFlow framework. Section 6.4 provides background about memory abstraction and information flow policies. Section 6.5 and Section 6.6 describe our demand-driven information flow mechanism. Section 6.7 discusses some practical limitations of DemandFlow, followed by the evaluations in Section 6.8.

6.2 Related Work

6.2.1 Static Information Flow Techniques

Language Based Techniques There has been a plenty of work in information flow tracking at compile-time for programs written in custom type-safe programming languages [125, 106]. Sabelfeld and Myers [125] present a comprehensive survey of this approach. This approach guarantees information flow security, but is limited to the programs written in specific languages. On the other hand, DemandFlow can

be applied to binary programs compiled from any language.

Source-code analysis Several tools have been suggested to track information-flow through source-code analysis. Livshits and Lam [101] propose a query language which can be used to represent taint-style vulnerabilities. Tripp et. al. [143] present an improved and scalable static taint analysis framework for JAVA programs utilizing an advanced pointer analysis and thin-slicing algorithm. Shankar et. al. [132] propose a type qualifier based approach for detecting format string vulnerabilities in C programs. Ashcraft et. al. [17] propose various compiler annotations and belief inference techniques to statically detect security holes in C programs.

Jovanovic et. al [86] present Pixy for statically detecting vulnerabilities in web applications. Xie et. al [158] statically expose vulnerabilities in scripting languages using precise information about memory locations. Balzarotti et. al [23] propose a framework to validate the functionality of taint sanitization functions in web applications.

All the above proposed approaches detect security vulnerabilities using source code and cannot be directly applied to executables. The precision of these approaches is driven by the underlying advanced pointer mechanisms, for which no counterpart mechanisms exist in executables, until now.

Executable-code analysis There has been a very limited amount of work on detecting information flow violations by statically analyzing the executable code. Major works in this approach are the vulnerability detection mechanism suggested by Cova et. al. [53], privacy leak detection [61] and integer flow vulnerabilities [155]. A major limitation of all these methods are that they ignore memory and aliasing is-

sues in their analysis, resulting in an imprecise vulnerability detection. As presented by Livshits and Lam [101], a precise points-to analysis is imperative for achieving a low false positive rate in any static framework.

An industrial tool, *Veracode* [13], uncovers vulnerabilities in executables. To the best of our knowledge, the techniques used by Veracode are proprietary and have not been published anywhere. Hence, the two could not be compared. Further, unlike DemandFlow, Veracode requires the presence of debug information, which is not present in deployed executables. A large number of other static executable analysis tools such as IDAPro [2], Divine [21], and several more, do not aim to uncover information flow vulnerabilities.

6.2.2 Dynamic Information Flow Techniques

There has been a large number of research tools for tracking information flow at runtime. Here, we only discuss some of the most popular and related dynamic techniques.

Schwartz et. al. [129] present an extensive survey discussing various dynamic taint mechanisms and their respective limitations. Newsome and Song [110] developed an early taint analysis mechanism to detect buffer overflow through runtime binary translation. However, they observed a huge slowdown of 37x. Several different approaches have been suggested to amortize the cost of taint propagation. Frameworks like LIFT [119], GIFT [94], Dytan [50] and tools proposed by Chang et. al. [43], Xu et. al. [160] and Jee et al. [85] aim to mitigate this overhead through

software techniques. TaintDroid [64] utilizes the optimizations present in the underlying virtual machine while Saxena et. al. [128] employ binary instrumentation to counter this overhead. As demonstrated by several frameworks [56, 147], in spite of these optimizations, dynamic methods still experience huge runtime overhead unless the tracking mechanism is implemented in hardware. On the other hand, DemandFlow detects vulnerabilities through static analysis and hence, does not result in any runtime overhead. Unlike our information abstraction which enables attribution, most of the dynamic frameworks employ single-bit taint tracking mechanisms. Maintaining an attribution mechanism in dynamic frameworks will further add to their overhead.

Several dynamic frameworks aim to mitigate their overhead using static schemes. However, such static schemes are completely different from DemandFlow. Frameworks like Chang et. al. [43] employ source code slicing mechanism to detect safe program locations while approaches like Xu et. al. [160] and Jee et al. [85] mitigate overhead by optimizing the actual taint instrumentation code in a local region. In contrast, DemandFlow presents a novel static framework on executables which computes fine-grained demand-driven program locations over the whole program.

Several systems have been proposed to apply information-flow analysis for detecting malicious software. For example, Panorama [163] and TaintDroid [64] provide a complete system-wide information flow system to distinguish malware and benign software. These methods propose new policies for detecting malicious activities while DemandFlow enables the enforcement of user-defined policies in applications.

Chang et. al. [43] were the first to recognize that taint tracking is a special case of data-flow analysis. However, they rely on availability of source-code and unlike DemandFlow, do not maintain an information abstraction.

6.2.3 Demand-driven Analysis

Demand-driven algorithms are popular in several compiler analyses. Heintze and Tardieu [79] present a demand driven approach for pointer analysis to tailor the computation to only a set of specific queries. Duesterwald et al. [59] propose demand-driven algorithms for interprocedural data-flow analysis. Our idea of demand-driven security analysis of executables is inspired by these compiler concepts.

Guyer et al. [75] propose a complementary client-driven approach of adapting the analysis to a particular set of queries. It dynamically varies the precision of analysis by employing flow-sensitivity or context-sensitivity depending on the requirement, but still computes an exhaustive solution. This idea can be combined with our demand-driven mechanism to obtain the benefits of both.

6.3 Overview of the system

Fig 6.1 presents an overview of the DemandFlow system. DemandFlow is built over the existing SecondWrite framework as presented in [10]. SecondWrite translates the input x86 binary code (including stripped executables) to the intermediate format of the LLVM Compiler [96]. SecondWrite implements various mechanisms to obtain an intermediate representation (IR) which contains features like procedure arguments,

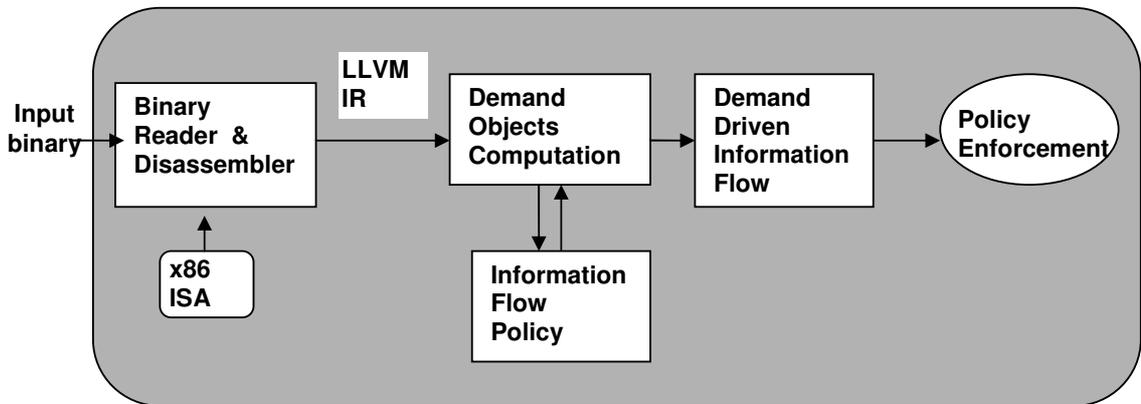


Figure 6.1: *Organization of the system.*

return values, types and high-level control flow. The recovered IR represents the symbols corresponding to the binary code’s registers in a single-static assignment (SSA) form. SSA form is widely used in many binary analyzers. Existing binary analysis tools implement various methods to recognize procedure arguments and procedure returns [21]. SecondWrite also implements various analyses to recognize the arguments. Once the arguments are recognized, formal arguments and returns are represented as a part of a procedure definition and actual arguments and returns are explicitly represented as a part of a call instruction in the IR.

This conversion to a compiler IR is not a necessity for our work. In fact, any existing static binary framework, for example those proposed by Balakrishnan and Reps [21] or Debray et al. [58], can also be employed for our analysis. Section 6.7 discusses various practical issues regarding the underlying binary framework. The LLVM IR obtained is passed through the demand-driven information flow framework.

An information flow policy, corresponding to each vulnerability, is formulated by specifying the initialization and enforcement conditions. Our Demand Object

Computation block in Fig 4.6 employs these initialization conditions to compute the required set of data objects (registers and memory locations) imperative for implementing the policy. Next, Demand-driven information flow analysis computes a set of information flow expressions only for the set of data objects computed by Demand Object Computation block. These information flow expressions are employed at the specified enforcement locations to detect policy violations. Multiple information flow violations can be detected by just specifying the initialization and enforcement conditions.

6.4 Background

6.4.1 Memory Abstraction

The memory abstraction for DemandFlow is defined by building on the concept of abstract memory regions and abstract locations (**a-locs**), defined by *Value Set Analysis* (VSA) [21]. VSA divides the address space of a program into several non-overlapping memory regions. Three kind of memory regions are defined: the set of memory regions associated with procedure stack frames in the program (**Stack**), the memory region associated with global data of the program (**Global**) and the memory regions associated with heap-allocations sites (**HeapRgn**). Each memory region is further abstracted through a set of **a-locs**. Intuitively, **a-locs** correspond to the program variables in each region. An **a-loc** is characterized by two attributes: its relative offset in the region with respect to other **a-locs** and its size.

Having defined **a-locs** as above, VSA computes an over-approximation of

the set of integers and the set of memory addresses (collectively referred to as a **Value-Set**) that each register and each `a-loc` holds at a particular program point. VSA employs affine relation and loop bound analysis to conservatively bound the memory locations accessed by any instruction. Hence, this algorithm can be used to determine the set of all possible memory locations referred to by all the *direct* and *indirect* memory access instructions. More details about this algorithm can be found in [21].

6.4.2 Information Flow Policy

Several information flow tracking systems express an information flow policy using the concept of labels [110, 119, 64]. Depending on the policy, labels can either refer to the input coming from an untrusted source or to an internal confidential information. There are four dimensions that characterize a policy: label description, label initializations, label propagation and label checks.

Label description, `LB`, specifies the underlying labels and optionally a union operator, `Operator`, governing merging of the labels at information join points in the program. *Label initializations*, `LBInit`, correspond to the sources which introduce labels into the program. *Label propagation*, governs the flow of these labels through the operations in a program. *Label checks*, `LBCheck`, denote the sensitive (or untrusted) program locations where an information flow violation might arise if an untrusted (or confidential) information reaches such locations.

The analysis framework, as presented in Sec 6.6, implements the underlying

label propagation mechanism. Consequently, an information flow policy is specified in our framework via the remaining three dimensions:

$$\text{Policy} \equiv \{ \{\text{LB}, \text{Operator}\}, \text{LBInit}, \text{LBCheck} \}$$

The above specification of information flow policy can also be defined using an information policy lattice \mathbf{I} with domain of values specified as a set $D_{\mathbf{I}}$ and a meet operator $\wedge_{\mathbf{I}}$, where $D_{\mathbf{I}}$ corresponds to the set of labels LB and $\wedge_{\mathbf{I}}$ corresponds to the meet operator Operator . This framework provides a generic and programmable framework for specifying several kinds of information flow policies such as format string vulnerability, directory traversal attacks or information leakage. We describe the policy specification using the example of format-string vulnerability.

Format string flaws arise due to an unsafe implementation of variable-argument procedures in C library. In case of a variable-argument procedure like `printf`, a `format string` argument specifies the number and type of other arguments. However, there is no runtime routine to verify that the procedure was actually called with the arguments specified by the `format string`. As detailed in [54], an attacker can corrupt the format string and thereby take control of the program by modifying relevant memory locations.

In order to expose a format string vulnerability, a tool needs to detect the flow of information from an untrusted source to the `format string` argument of a variable argument procedure. All program inputs that can be controlled by an attacker are treated as `tainted` values and the `tainted` information is propagated

through the program. The presence of a `tainted` value in the format string argument of any variable argument procedure signifies the presence of a vulnerability. This can be formally described as

$$\begin{aligned}
 D_I(\text{LB}) &:= \{\text{tainted}, \text{untainted}\} \\
 \wedge_I(\text{Operator}) &:= \{\text{tainted} \wedge_I \text{untainted} \equiv \text{tainted}\}
 \end{aligned} \tag{6.1}$$

6.5 Demand Driven Set

In an exhaustive analysis, the information is propagated over the complete flow of a program. However, such an exhaustive analysis might implicitly propagate information along data objects which do not impact the current policy enforcement decision. Recall from Section 6.1, this redundant propagation of information flow in a program limits the scalability of an analysis and forces the analysis to make imprecise decision to maintain scalability.

Next, we present our analysis to compute the set of data objects necessary to enforce a particular information flow policy. This required set of data objects (variables and memory locations) is represented as **Demand Set**.

$$\text{Demand Set} = \begin{cases} \text{SR} : \textit{Set of required variables} \\ \text{SM} : \textit{Set of required a-locs} \end{cases}$$

Sets **SR** and **SM** are collectively referred to as **Demand Set**. We refer to an element of set **Demand Set** as a **Demand Object**. Set **SM** consists of **a-locs** from all three memory regions **Stack**, **Global** and **HeapRgn** mentioned in Section 6.4.1. We

employ the logical inference form¹ for representing the deduction rules for computing the sets **SR** and **SM**.

Fig 6.2 presents the rules for computing **Demand Set**. The rules are applicable to operations in the IR, but we present C-like pseudo instructions for ease of understanding. The rules constitute a backward analysis, where the instructions are traversed in a demand-driven backward dataflow order.

At the beginning of the analysis, set **SM** is initialized as an empty set while set **SR** is initialized with the variables employed at program locations of possible information flow violations, which includes all elements in set **LBCheck**. For example, in case of a format string vulnerability, set **LBCheck** comprises format string arguments at all the format string callsites in a program.

Given an initial set of elements in **Demand Set**, the rules presented in Fig 6.2 analyze each program operation to update **Demand Set** accordingly. In case of an assignment operation, if the destination is already a **Demand Object**, the source operand is also added to the set. In case of arithmetic and logical operations, both the source operands are added to set **SR**, if it already contains the destination.

The memory load and store operations employ Value Set Analysis (VSA) [21] to update **Demand Set**. In case of a memory load operation, the **a-locs** present in the value set of the source operand are added to set **SM** only if the loaded value is already a **Demand Object**. Similarly, a value employed in a memory store operation is considered a **Demand Object**, if any of the possibly accessed **a-loc** is an element

¹The expression $\frac{\text{Premise \#1} \quad \text{Premise \#2} \quad \dots \quad \text{Premise \#n}}{\text{Conclusion}}$ states that whenever the given set of premises have been obtained, the specified conclusion can be taken for granted as well.

Helper Variables

VS(R): Value Set of object R; $R \rightarrow z : \text{a-loc } z \in \text{VS}(R)$

OP : Arithmetic, Logical and Casting operators

ARG_T : Set of parameters of procedure T

RET_T : Set of variables at actual return-sites in procedure T

FORM_i : Variable for *i*th formal parameter of a procedure

ACT_i : Variable for *i*th actual parameter at a callsite

F: An internal procedure; X: An external procedure

Initialization

SR \leftarrow LBCheck; SM \leftarrow { }

Rules

$$I: R1=R2 \quad \frac{R1 \in \text{SR}}{R2 \in \text{SR}}$$

$$I: R1=R2 \quad \text{OP} \quad R3 \quad \left\{ \begin{array}{l} \frac{R1 \in \text{SR}}{R2 \in \text{SR}} \quad \frac{R1 \in \text{SR}}{R3 \in \text{SR}} \end{array} \right.$$

$$I: R1=*R2 \quad \frac{R1 \in \text{SR} \quad R2 \rightarrow z}{z \in \text{SM}}$$

$$I: *R1=R2 \quad \frac{R1 \rightarrow z \quad z \in \text{SM}}{R2 \in \text{SR}}$$

$$I: R1=\text{call } F \quad \left\{ \begin{array}{l} \frac{\forall_{i \in \text{ARG}_F} \frac{\text{FORM}_i \in \text{SR}}{\text{ACT}_i \in \text{SR}}}{\forall_{i \in \text{RET}_F} \frac{R1 \in \text{SR}}{i \in \text{SR}}} \end{array} \right.$$

$$I: R1=\text{call } X \quad \left\{ \begin{array}{l} \frac{\forall_{i \in \text{ARG}_X} \frac{R1 \in \text{SR}}{\text{ACT}_i \in \text{SR}}}{\forall_{i \in \text{ARG}_X} \frac{R1 \in \text{SR} \quad \text{ACT}_i \rightarrow z}{z \in \text{SM}}} \end{array} \right.$$

Figure 6.2: *Deduction rules for computing Demand Set. Rules constitute a backward analysis, where a conclusion before an instruction is derived based on the premise after the instruction.*

of set SM.

Interprocedural rules in Fig 6.2 depend on whether the called procedure is an internal or an external procedure. The distinction is required due to the absence of procedure body of externally called procedures. In case of a call to an internal procedure, an **actual** argument value at the call site is added to set SR if the corresponding **formal** argument to the procedure is already present in set SR. A return value also results in a similar update of set SR. If the **actual** return value at

the call-site is present in `SR`, then all the return variables in the procedure definition are also considered as `Demand Objects`.

A call to an external procedure is handled in one of the following two ways. If the prototype of the called procedure is available, then the call is modeled by adding all actual arguments and their underlying `a-locs` to `Demand Set` if the return value is a `Demand Object`. Otherwise, a call to a procedure with unknown prototype is modeled as `NOP` to avoid excessive loss of precision.

The `Demand Set`, comprising `SR` and `SM`, captures all the variables and memory locations which can possibly impact the value of the elements in set `LBCheck`. This reduced set is employed to compute the information flow in the program.

6.6 Demand Driven Information Flow Analysis

Our demand-driven information flow analysis, *Symbolic Information Analysis*, is a flow-sensitive, context insensitive analysis which computes a conservative over-approximation of a set of sources of information reaching each demand driven data object (variables and memory locations) at each program point. This analysis employs `Demand Set` computed through the mechanism in Section 6.5.

6.6.1 Information Abstraction

Symbolic information analysis represents the values in an abstract domain defined by the `Symbolic Information Grammar` presented in Fig 6.3. The sentences generated by the `Symbolic Information Grammar` constitute the underlying symbolic

$INF := INF \cup TjT$ $T := [IR\ Symbols]$
--

Figure 6.3: *Symbolic Information Grammar: Grammar for information flow abstraction.* \cup is the union operator, *IR Symbols* are symbols in the obtained intermediate representation corresponding to the registers in the input executable, intermediate computations and calls to external library procedures.

information abstraction. An element of this grammar represents an object in our abstract domain and is represented as `SymInf`.

`SymInf = An element of Symbolic Information Grammar`

The analysis computes a `SymInf` object for each element of `Demand Set`. As evident from Fig 6.3, each `SymInf` object is a logical union of symbols in the intermediate representation (IR). `IR symbols` comprise the symbols in the intermediate representation corresponding to calls to external library procedures, local computations or any other information source such as a protected file or a secure memory location. `SymInf` abstraction captures a conservative over-approximation of the set of sources from which information can flow to a particular element of `Demand Set`.

`SymInf` abstraction has two advantages over the standard single bit taint abstraction. First, this abstraction enables the *attribution* of each policy violation to the corresponding culprit information source or a set of sources. There can be multiple sources of external information in a program and some of these sources might not result in a violation. The ability to attribute a violation to the actual information source is imperative if the framework is employed for rectifying the violations, in addition to the detection of violations. Second, this abstraction efficiently solves

the challenge of *time of detection/time of attack gap* [129] faced by several existing information flow frameworks [137]. Single-bit taint analysis raises a warning when tainted values are used in an unsafe manner. However, there is no guarantee that the program integrity has not been violated before this point. Several frameworks such as BitBlaze [137] circumvent the problem of too-little taint information by performing post hoc instruction trace analysis on the execution traces to determine the time of attack. **SymInf** abstraction obviates the need of any such post hoc analysis.

6.6.2 Analysis

The analysis computes flow-sensitive **SymInf** abstraction for all elements of **Demand Set** with respect to a particular information flow policy. A flow-sensitive analysis needs to compute the abstraction at each program point. Since we assume that the IR supports SSA form for variables, a single symbolic map is sufficient to maintain flow-sensitive **SymInf** abstraction for variables. Since memory locations are usually not implemented in SSA format, a map is maintained at each program point to represent flow-sensitive abstraction for memory locations. Hence, the analysis effectively computes the following maps, which collectively constitute the information flow abstraction for **Demand Set**.

IR: **SymInf** for elements in **SR**

IM_e: **SymInf** for elements in **SM** after program point *e*

Fig 6.4 presents the rules for computing the abstraction. The rules presented in Fig 6.2 compute **Demand Set** using a backward propagation mechanism while the

rules presented in Fig 6.4 forward propagate the information abstraction over the elements of `Demand Set`.

These rules analyze each program operation to update the information abstraction maps accordingly. In case of assignment and arithmetic operations, the abstraction is computed for the destination only if the source operands are present in `Demand Set`. The information flow abstraction is represented by the union (\cup) of abstract values of individual source operands. Analogous to the rules in Fig 6.2, VSA is employed to compute the abstract values for memory load and store operations. In case of a memory load operation, the value is computed by unioning abstract values of all the individual `a-locs` possibly accessed by this operation. Similarly, a memory store operation is modeled by updating the value of all possibly accessed `a-locs` with abstract values of the stored operand. As per standard compiler representation, this corresponds to *weak-updates*.

A call to an internal procedure is handled by forward propagating the `SymInf` abstraction from an actual argument at the callsite to the corresponding formal argument of procedure definition when both the arguments are `Demand Objects`. Similarly, the value for the required return variables at a callsite is computed using corresponding actual return values in procedure body.

A call to an externally called procedure is handled in a conservative manner. This rule models the flow of information from all sources available to this call to all possible destinations. As evident from the Fig 6.4, this rule also results in addition of a new information source, `IR symbol` of called procedure, to `SymInf` abstraction.

First, all the possible sources of information to this particular call are deter-

Helper Variables

$\overline{VS}(R)$: Value Set of object R; $R \rightarrow z : \text{a-loc } z \in \overline{VS}(R)$

IM'_e : SymInf for SM before program point e

ARG_T : Set of parameters of procedure T

RET_T : Set of variables at actual return-sites in procedure T

FORM_i : Variable for ith formal parameter of a procedure

ACT_i : Variable for ith actual parameter at a callsite

F: An internal procedure; X: An external procedure

Initialization

$\text{IR} \leftarrow \{ \}; \text{IM}_e \leftarrow \{ \} \forall e$

Rules

$$\begin{array}{l}
 \text{I: } R1=R2 \quad \frac{R1 \in \text{SR} \quad R2 \in \text{SR}}{\text{IR}(R1) \leftarrow \text{IR}(R2)} \\
 \\
 \text{I: } R1=R2 \text{ OP } R3 \quad \left\{ \begin{array}{l} \frac{R1 \in \text{SR} \quad R2 \in \text{SR}}{\text{IR}(R1) \leftarrow \text{IR}(R1) \cup \text{IR}(R2)} \\ \frac{R1 \in \text{SR} \quad R3 \in \text{SR}}{\text{IR}(R1) \leftarrow \text{IR}(R1) \cup \text{IR}(R3)} \end{array} \right. \\
 \\
 \text{I: } R1=*R2 \quad \frac{R1 \in \text{SR} \quad R2 \rightarrow z \quad z \in \text{SM}}{\text{IR}(R1) \leftarrow \text{IR}(R1) \cup \text{IM}'_I(z)} \\
 \\
 \text{I: } *R1=R2 \quad \frac{R1 \rightarrow z \quad z \in \text{SM} \quad R2 \in \text{SR}}{\text{IM}_I(z) \leftarrow \text{IM}_I(z) \cup \text{IR}(R2)} \\
 \\
 \text{I: } R1=\text{call } F \quad \left\{ \begin{array}{l} \frac{\forall i \in \text{ARG}_F \quad \text{FORM}_i \in \text{SR} \quad \text{ACT}_i \in \text{SR}}{\text{IR}(\text{FORM}_i) \leftarrow \text{IR}(\text{FORM}_i) \cup \text{IR}(\text{ACT}_i)} \\ \frac{\forall i \in \text{RET}_F \quad R1 \in \text{SR} \quad i \in \text{SR}}{\text{IR}(R1) \leftarrow \text{IR}(R1) \cup \text{IR}(i)} \end{array} \right. \\
 \\
 \text{I: } R1=\text{call } X \quad \left\{ \begin{array}{l} \text{let } \text{TMP} = \bigcup_{i \in \text{ARG}_X} \{ \text{IR}(\text{ACT}_i) \cup_{z \in \text{VS}(\text{ACT}_i)} \text{IM}'_I(z) \} \\ \frac{\forall i \in \text{ARG}_X \quad R1 \in \text{SR} \quad \text{ACT}_i \in \text{SR}}{\text{IR}(R1) \leftarrow \text{IR}(R1) \cup \text{TMP} \cup X} \\ \frac{\forall i \in \text{ARG}_X \quad R1 \in \text{SR} \quad \text{ACT}_i \in \text{SR} \quad \text{ACT}_i \rightarrow z \quad z \in \text{SM}}{\text{IM}_I(z) \leftarrow \text{IM}_I(z) \cup \text{TMP} \cup X} \end{array} \right.
 \end{array}$$

Figure 6.4: *Rules for Symbolic Information Analysis.*

mined. This includes the actual argument values as well as `a-locs` accessed by these arguments. Next, the values of return variable as well as all possibly accessed `a-locs` are updated to reflect the flow of information from all these sources. The `IR symbol` corresponding to the actual called procedure is also added to the abstract values of return as well as above `a-locs`. The precision of analysis can be improved by adding the actual semantic model corresponding to popular external procedures such as `sprintf`.

Our analysis handles well-known information cancellation idioms like `xor` a value with itself [129]. Fig 6.4 handles information flow for all arithmetic, logical or casting operators. The negation operator is implicitly handled using the assignment rule. In order to limit the exponential growth of information sources, a widening operator is employed to impose a limit on the cardinality of each `SymInf` object, defined as `L`, at the cost of some precision. `L`, was kept to 20 in our framework.

6.6.3 Policy Enforcement

`SymInf` abstraction can be represented using a lattice framework, referred to as *symbolic lattice* \mathbb{V} . We introduce the following definitions to aid the understanding:

S : Set of all sentences generated by Symbolic Information Grammar (Fig 6.3)

S_L : Subset of S with cardinality limit of L , $\bigcup_{p \in S} \{p \mid \|p\| \leq L\}$

S'_L : $S_L \cup S$

The domain of values of lattice V is the modified set of sentences, S'_L , described above. The empty set ϕ represents the unique largest element, \top , and set S is the unique smallest element, \perp of the lattice. Lattice V , can mathematically be represented as follows:

$$\begin{aligned} D_V &:= S'_L \\ \wedge_V &:= \cup \end{aligned} \tag{6.2}$$

Section 6.4.2 formulated an information flow policy through a lattice $I = \{D_I, \wedge_I\}$, referred to as *policy lattice*. The *symbolic lattice* described above can be mapped to obtain the *policy lattice*. The labels derived by the resulting policy lattice can be employed at the program locations given by `LBCheck` (Section 6.4.2) to detect a violation.

Given a *policy lattice* I , let us define a function ϕ_I which maps the domain of values in *symbolic lattice* V to the domain of values in lattice I .

$$\phi_I : D_V \rightarrow D_I \tag{6.3}$$

Each element, S , of set D_V is a sentence generated by the grammar in Fig 6.3. Each sentence, in turn, comprises of `IR symbols` (terminal symbols of this grammar). Let $T(S)$ denote the set of `IR symbols` in a sentence S .

`LBInit` (Section 6.4.2) specifies the sources through which the information flow labels enter the program. For example, `IR symbols` corresponding to the library

procedures that introduce untrusted values in the program. As per the rules in Fig 6.4, these IR symbols are part of `SymInf` abstraction computed by the analysis.

IR symbols in a program can be divided in two categories based on whether `LBInit` maps such symbols to a label or not. Based on these categories, a function `LBMAP`, mapping IR symbols to the domain of values in policy lattice, can then be defined as follows:

```

LBMAP(r) :if LBInit.hasEntry(r)
    return LBInit(r)
else
    return  $\top$ 

```

(6.4)

If IR symbol `r` has an entry in `LBInit`, `LBMAP` returns the corresponding label, otherwise it returns the element \top in lattice `I`. For example, in the case of format string vulnerability, the external procedures that do not represent any information source are mapped to the lattice element `untainted`. This does not impact the precision of the analysis, since the relation $\top \wedge \mathbf{x} = \mathbf{x}$ holds true in any lattice.

Having defined `LBMAP` as above, we can define function ϕ_I . It maps each constituting IR symbol in `Dv` to a lattice element in `DI` and merges the lattice elements using the meet operator of lattice `I`. Mathematically, this function can be defined as follows:

$$\phi_I(S) : \bigwedge_{t \in T(S)} \{LBMAP(t)\} \tag{6.5}$$

Henceforth, a `tainted` label at any location specified in `LBCheck` signifies a vulnerability. For example, in case of format string vulnerability, `tainted` label for format string arguments flags a warning. Given an information flow policy, we can always define a corresponding function ϕ_I which maps the symbolic lattice to policy lattice. The existence of such a mapping provides an extensible and generic framework for specifying any information flow policy.

As presented by Chang et al. [43], single-bit taint frameworks cannot expose several vulnerabilities, like file-disclosure vulnerabilities, which require multiple-bit information to be tracked simultaneously. The mapping presented above enables DemandFlow to reliably expose such vulnerabilities.

6.7 Discussion

We now consider some practical issues of DemandFlow.

6.7.1 Indirect calls and branches

The underlying binary system employed for DemandFlow, SecondWrite, implements various mechanisms [135] to address code discovery problems and to handle indirect control transfers. Here, we briefly summarize their mechanism.

A key challenge in binary frameworks is discovering which portions of the code section in an input executable are definitely code. Smithson et al. [135] proposed *speculative disassembly*, coupled with *binary characterization*, to efficiently address this problem. SecondWrite speculatively disassembles the unknown portions of the

code segments as if they are code. However, it also retains the unchanged code segments in the IR to guarantee the correctness of data references in case the disassembled region was actually data.

SecondWrite employs *binary characterization* to limit such unknown portions of code. It leverages the restriction that an indirect control transfer instruction (CTI) requires an absolute address operand, and that these address operands must appear within the code or data segments. The code and data segments are scanned for values that lie within the range of code segment. The resulting values are guaranteed to contain, at a minimum, all of the indirect CTI targets.

The indirect CTIs are handled by appropriately translating the original target to the corresponding location in IR through a translator procedure. Each recognized procedure (through speculative disassembly) is initially considered a possible target of the translator, which is pruned further using alias analysis. The arguments for each possible target procedure are unioned to find the set of arguments to be passed to the translator; a stub inside the translator populates the arguments according to the actual target.

This method is not sufficient for discovering indirect branch targets where addresses are calculated in binary. Hence, various procedure boundary determination techniques, like ending the boundary at beginning of next procedure, are also proposed [135] to limit the possible targets.

The above mechanism of handling indirect control transfers is not a necessity for DemandFlow; methods suggested by any other binary framework [21, 58] can also be employed.

6.7.2 Limitations

Here, we present some limitations of DemandFlow.

Implicit flows As evident from Fig 6.4, DemandFlow only performs *explicit information flow*, that is information-flow based on data computations, and is not capable of detecting vulnerabilities or illegal flows arising due to *implicit information flow*, that is program’s flow of control. As is well accepted by the community [92], handling implicit flows results in a large number of false positives. Hence, most of the practical static information flow tools, excluding the tools that enforce non-interference through language-based techniques, ignore implicit flows. In the future, we plan to expand DemandFlow to handle implicit flows also.

Limitations of static executable analysis As discussed at a recent Dagstuhl Seminar [91], static analysis of executables provides a variety of benefits over dynamic mechanisms. However, several executable artifacts like indirect calls pose a significant challenge to the scalability of sound static analyses. It was decided that the verification of a browser is a laudable long-term goal of static executable analyses.

Since DemandFlow is based on static analysis of executables, our evaluation is limited to the applications which can be reliably handled through any static mechanism. Our theoretical frameworks have no inherent scalability limitations. The evaluations are presented for large server and client programs like *apache*, *lynx* and *MySQL*; continuous progress in improving the scalability of static techniques in general will broaden the application of our framework to even larger applications

like *Chrome*.

Dynamically generated code Static analysis tools have a limitation that the code analyzed might not be the code which actually executes. A small percentage of programs include self-modifying code, few packed executables unpack themselves at runtime and browsers like *Chrome* employ just-in time compilation mechanism to dynamically generate a portion of code. It is impossible for any analysis tool to statically reason about the code generated at runtime. Various methods [156] statically detect the presence of runtime code generation in a program. Such a tool can be integrated in our front-end, to at least warn the user.

Obfuscated code SecondWrite has not been tested against binaries with hand-coded assembly or with obfuscated control flow. We will investigate such programs in the future.

6.8 Results

In this section, we evaluate DemandFlow on a set of real-world programs listed in Fig 6.5 and a wide set of compute-bound programs including the complete *SPEC2006 benchmarks* suite. DemandFlow’s versatility is demonstrated by extending the underlying analysis for three different information flow violations - format string vulnerability, directory traversal attack and information flow leakage. Section 6.8.1, 6.8.2 and 6.8.3 discuss the uncovered vulnerabilities, false alarms statistics and scalability for these programs respectively. Since SPEC2006 benchmarks have no known vulnerabilities, their false alarms aspects are discussed separately in

Application	LOC	Vulnerability	Type
mingetty 1.08	500	-	-
csplit 8.17	1,060	NEW	Format String
muh 2.05c	2857	CVE-2000-0857	Format String
pfingerd 0.7.8	4689	NISR16122002B	Format String
gzip 1.2.4	5830	CVE-2005-1228	Directory Trav
ez-ipup3.0.10	6,335	CVE-2004-0980	Format String
gif2png 2.5.2	9354	CVE-2010-4695	Directory Trav
wu-ftpd 2.6.0	17576	CVE-2000-0573	Format String
tar 1.13.19	20518	CVE-2001-1267	Directory Trav
KeePassX0.4.3	26089	-	-
yafc 1.1.1	32,241	NEW	Directory Trav
tnftp 2010	34,762	-	-
gftp 2.0.19	42,390	-	-
irc2 2011	44,837	NEW	Directory Trav
wget 1.13	46,611	-	-
sudo 1.8	53,144	CVE-2012-0809	Format String
openssh 6.0p	73335	-	-
opensshd 6.0p	73335	-	-
ayttm 0.6.3	80,013	NEW	Format String
curl 7.30.0	122,248	NEW	Directory Trav
BitchX 1.1	133,728	NEW	Format String
lynx 2.8.7	135,876	-	-
apache 2.2.17	232,778	-	-
MySQL 5.6.11	1,741,774	-	-

Figure 6.5: *Vulnerabilities discovered in real-world programs.*

Appendix 6.8.5. Section 6.8.4 highlights DemandFlow’s *extensibility* by extending the framework for detecting information-flow leakage. The programs are compiled with gcc v4.4.1 without any symbolic or debug information. Results are obtained on a 2.4GHz 8-core Intel Nehalem machine running Ubuntu. The underlying disassembly mechanism (Section 6.7.1) employed in SecondWrite results in 100% code coverage in the above set of programs.

DemandFlow is highly scalable and analyzes each of the the programs in Fig 6.5 in less than a minute, except MySQL (1.7 million LOC) which took seven minutes.

The results on compute intensive SPEC benchmarks (Appendix 6.8.5) demonstrate a moderate storage requirement of under 100 MB.

Fig 6.5 includes commonly-used server programs (*pfingerd*, *muh*, *wu-ftpd*, *openssh*, *apache*), popular client programs (*ez-update*, *yafc*, *tnftp*, *gftp*, *wget*, *openssh*, *curl*, *lynx*), internet relay chat clients (*irc2*, *ayttm*, *BitchX*) and several utility programs (*mingetty*, *gif2png*, *csplit*, *tar*, *gzip*, *sudo*, *KeePassX*). These are widely deployed applications and their integrity is essential for a smooth functioning of the system.

6.8.1 Vulnerabilities

In this section, we discuss DemandFlow’s ability to uncover standard vulnerabilities such as format string and directory traversal attacks. As explained in Section 6.4.2, format string flaws arise due to an unsafe implementation of variable-argument procedures in C. A directory traversal vulnerability typically arises when a filename supplied by an user is employed in a file-access procedure without sufficient validation. A malicious user can malform the name by including a *..* (*dot dot*) within the response, thereby gaining an ability to ascend outside the authorized directory. This vulnerability can be uncovered in a similar manner, by assigning a *tainted* label to the inputs coming from an untrusted channel and raising an alarm at any use of a *tainted* value as a filename argument to any file-access function.

Fig 6.5 shows that DemandFlow uncovers six previously unknown vulnerabilities, apart from detecting all previously known vulnerabilities in this set of programs. Next, we discuss the characteristics of these zero-day vulnerabilities.

<pre>static char * suffix ; main:: switch (..) { case 'b': //Unsafe Initialization suffix = optarg; break;} make_filename: sprintf(filename_space, suffix, //Format string Arg num);</pre>	<pre>0x8056160: Fixed location for optarg; 0x80561ac: Memory address of suffix (Address) (Instruction) main: 804afb4: mov 0x8056160,%eax //Load from optarg 804afb9: mov %eax,0x80561ac //Store to suffix 804afbe: jmp 804b0b6 <main+0x1f8> make_filename: 804a18e: mov 0x80561ac,%eax //Load from suffix 804a1c6: mov %eax,0x4(%esp) //Initialize format arg 804a1ca: mov %edx,(%esp) 804a1cd: call 8048ec0 <sprintf@plt></pre>
(a) Source code snippet	(b) Executable code snippet

Figure 6.6: Code snipped from *csplit* showing the format string vulnerability. Second operand is the destination in executable code.

csplit: *csplit* is a well-known GNU *Coreutil* program. DemandFlow detected a possible format string vulnerability in this utility. Fig 6.6 shows the corresponding source-code snippet as well as executable code snippet. *csplit* declares a global variable `suffix`, which is initialized in procedure `main` using an input argument (`optarg`). Next, `suffix` is employed directly as a format string argument in a call to `sprintf`. DemandFlow flagged this unsafe information flow from an external source to a format string argument. We notified the *Coreutil* developers about this vulnerability. They pointed to an implicit sanitization procedure, but are validating its behavior for this new vulnerability.

Fig 6.6(b), the global variable `suffix` is allocated to a memory location in the executable. DemandFlow would not have uncovered this unsafe flow if the information is not propagated across memory locations. This example underscores the importance of our precise memory analysis for exposing information flow vulnera-

bilities in executables.

ayttm: *ayttm* is vulnerable to a previously unknown format string attack. In *ayttm*, a procedure `http_connect` populates a variable `inputline` by receiving data from network using a call to external procedure `recv`. Then, `inputline` is assigned to a variable `debug_buff` using `snprintf`, which is further used as a format string argument in a `printf` call. This vulnerability has been confirmed by the developers.

BitchX: DemandFlow exposes a format string vulnerability in *napster* plugin in BitchX. The behavior is similar to the vulnerability in *csplit*, where an input argument value is employed as a format string argument in a call to `vsnprintf`.

yafc, *irc2*, *curl*: DemandFlow exposes directory traversal vulnerabilities in each of these three programs. The underlying behavior of the uncovered vulnerabilities is similar in all these programs. These programs employ `getenv` to derive the name of the current directory and prepend the resulting value to derive the name of a file. This resulting filename is employed to open a file using a `fopen` call without any sanitization. As per several existing attacks [1], an attacker might corrupt the environment variables, Hence, employing environment variables for deriving a filename renders the application susceptible to directory traversal attacks. We have notified the respective developers.

In all these programs, maintaining the actual information source as part of SymInf abstraction, instead of a single bit taint information, directly exposes the corresponding unsafe source without any post-hoc analysis. This eases the task of understanding the behavior of the uncovered vulnerabilities.

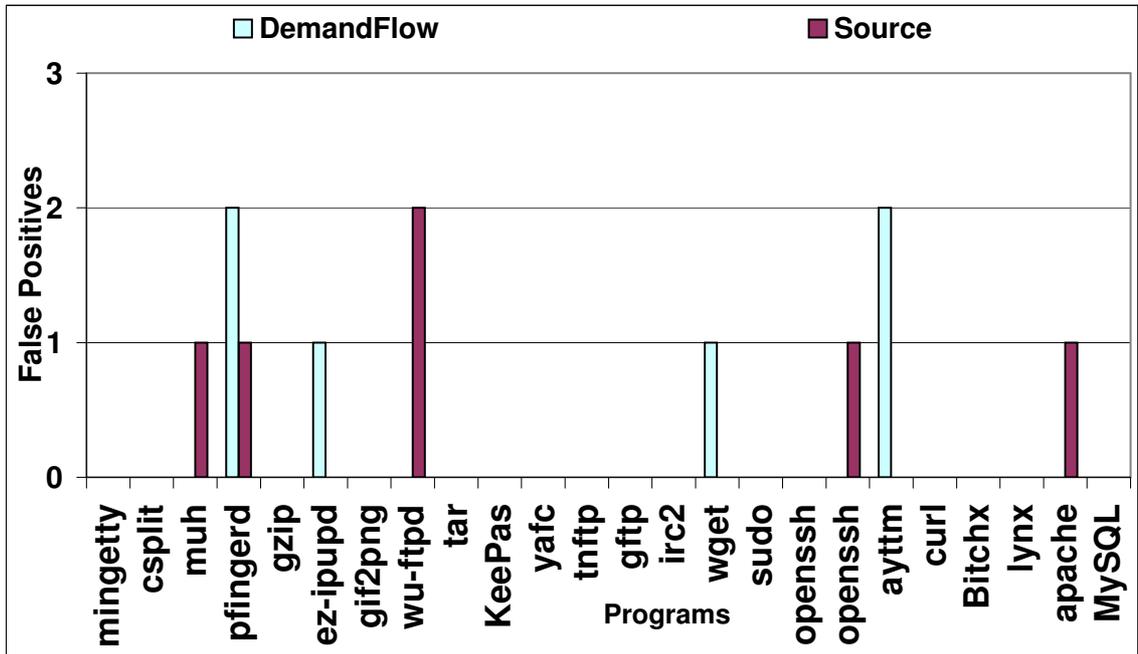


Figure 6.7: *Format string vulnerability detection.*

Next, we establish the importance of reasoning about memory accesses for vulnerability detection. In order to simulate the functionality of previous tools [53], which do not track memory locations, the analysis presented in Section 6.6 is modified to compute SymInf abstraction for only the variables. This is accomplished by disabling the rules in Fig 6.4 for memory access instructions and by computing only IR. *The resulting analysis fails to unmask even a single vulnerability in the programs listed in Fig 6.5.* This demonstrates the importance of a precise memory analysis for implementing a robust information flow mechanism in executables.

6.8.2 False Positives

Fig 6.7 presents the false positives reported by DemandFlow for each of the programs in Fig 6.5 while detecting format string vulnerabilities, comparing the resulting

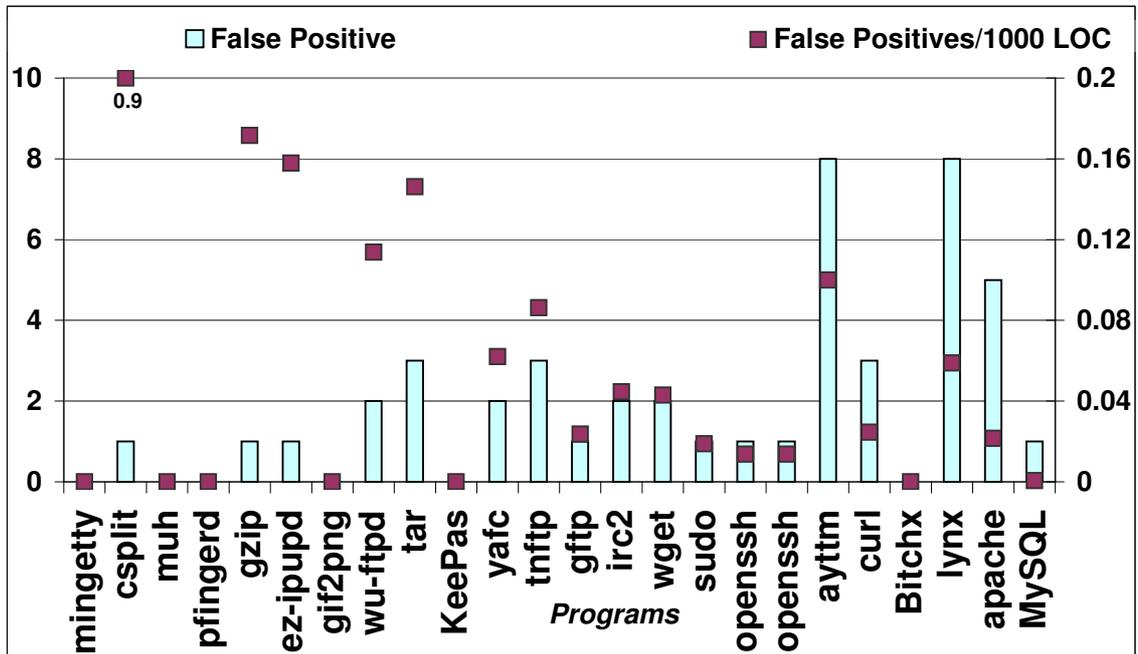


Figure 6.8: *Directory traversal attacks.*

statistics with the false positive reports generated by existing source-level static analysis tools (Oink [4], CQual [132] and others [75]) we ran against the same programs². DemandFlow reports similar false alarms as existing source-level tools for the programs listed in Fig 6.5.

Fig 6.8 presents the corresponding statistics obtained for the directory traversal vulnerability. Even though DemandFlow reports eight false positives for *lynx* and *aytm*, it translates to less than 0.1 false alarms per 1000 lines of code. To the best of our knowledge, no existing source-level static analysis tool has reported directory traversal vulnerability statistics for the above set of programs, consequently, the results could not be compared.

The false positive rate (FP/Total Reports) is 79.1% for above programs which

²The programs with no corresponding results by source-tools are conservatively assumed to have zero false positives.

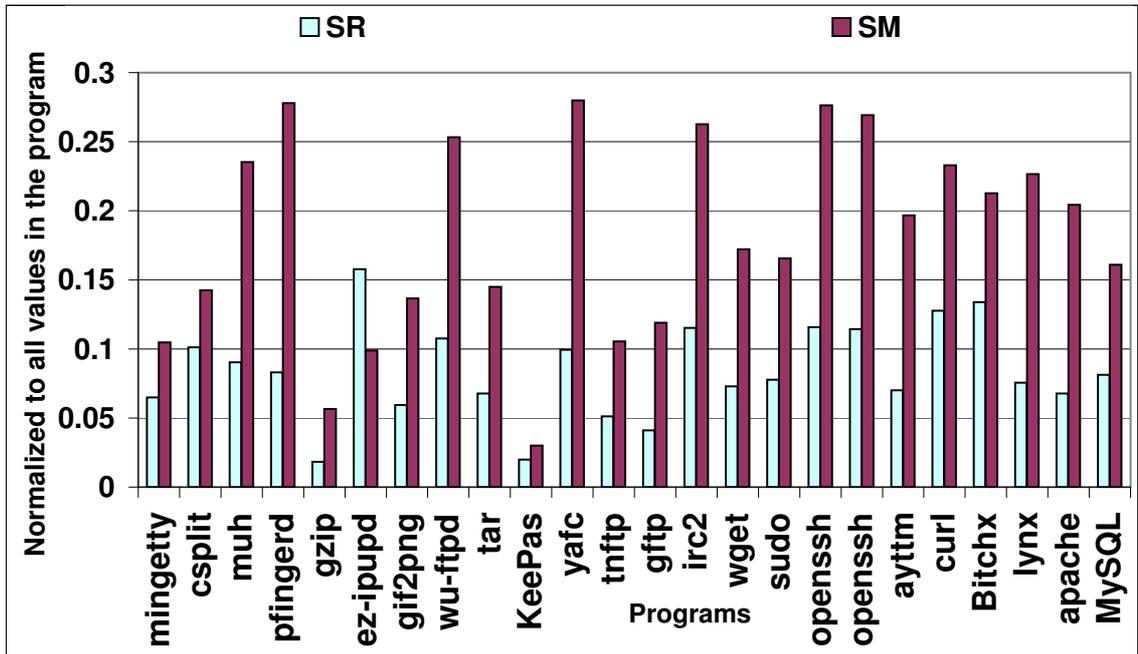


Figure 6.9: *Size of Demand Set (SR and SM) normalized (=1.0) to all variables and a-locs respectively.*

is slightly better than 84% false positive rate [43] reported by source-tools like Oink and CQual [4, 132]. In total, only around 50 false positives were reported in programs coming from more than 5 million lines of code. Corresponding statistics for SPEC benchmarks are presented in Appendix 6.8.5.

6.8.3 Scalability

Recall from Section 6.3, our demand driven analysis enhances DemandFlow’s scalability. Here, we quantify this enhancement.

Fig 6.9 presents the size of Demand Set, SR and SM, as determined using the rules in Fig 6.2, for detecting format string vulnerability. The sizes of SR and SM are normalized against the total number of variables and a-locs in the program respectively. Fig 6.9 shows that these rules are highly efficient in decreasing the

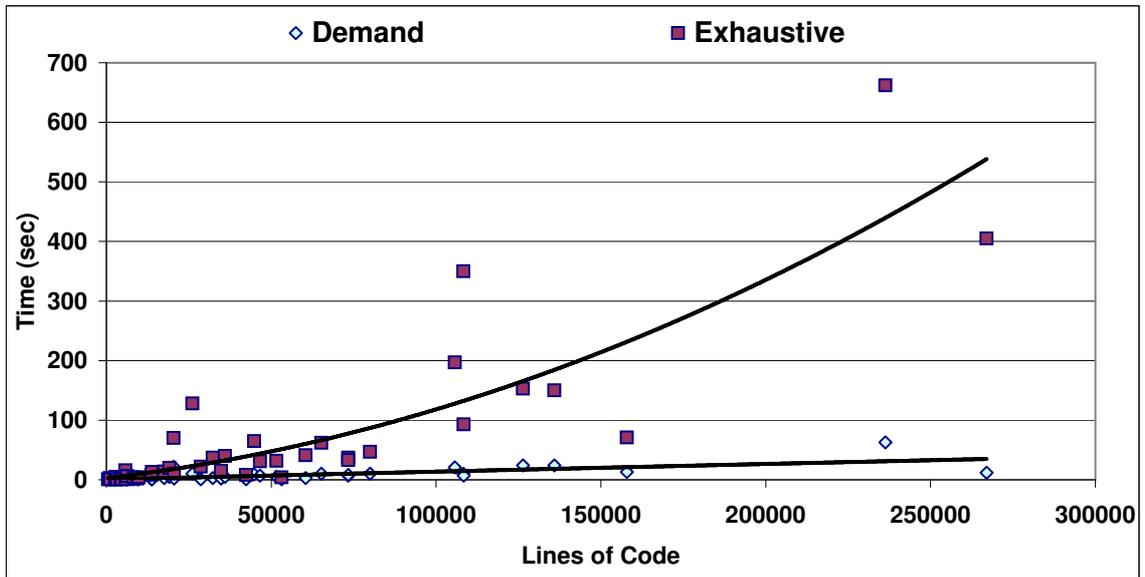


Figure 6.10: Scalability of demand driven and exhaustive analysis with increasing lines of code.

overall analysis requirement. This enables DemandFlow to only analyze around 20% of total objects, on average, without sacrificing the precision. *KeypassX*, being a C++ program, does not have many `format string` calls. Hence, it has relatively small SR and SM set.

Fig 6.10 highlights the ensuing enhancement in DemandFlow’s scalability as a result of employing a Demand Set. It plots the variation in the time taken to analyze the programs in a demand-driven manner with increasing lines of code and compares it with an exhaustive analysis over the complete program. The exhaustive analysis is implemented by discarding the Demand Set and applying the rules in Fig 6.4 for all program objects. Fig 6.10 includes the programs listed in Fig 6.5 as well as programs from complete *SPEC2006* benchmark suite. As evident, demand-driven analysis is approximately 10x more scalable than the exhaustive analysis. For example, the time to analyze *gcc*, a large SPEC2006 benchmark with 250,000

lines of code, reduces to less than a minute as compared to more than 11 minutes in exhaustive analysis. This scalability becomes more evident in programs like *MySQL* where demand mechanism was able to finish the analysis in 7 minutes (not shown in the graph) as compared to more than an hour of exhaustive analysis.

6.8.4 Information Flow Leakage

6.8.4.1 KeePassX

We employ DemandFlow to understand the flow of password information in *KeePassX* [3], a popular open-source password manager utility. It stores the passwords in an encrypted database, protected by a master password.

KeePassX decrypts the stored passwords using a special *unlock* procedure. The callsites to procedure *unlock* are marked as *confidential* locations while all the unknown external procedure callsites (*writing to file, mapping with keyboard symbol, console output*) are marked as *untrusted* locations. The resulting analysis reveals various program points where *confidential* information flows into *untrusted* channels. These locations include methods for keyboard symbol conversion for auto-typing the password to a desired location and writing to a file for exporting the databases.

The information flow for auto-typing a password involves possibly unsafe operations. The auto-typing to a desired location is accomplished through a hot-key mechanism. On pressing the hot key, the utility looks up the correct entry in the database and executes its auto-type sequence. However, the manual analysis of the code revealed that *KeePassX* only compares the title of the current window

while searching for the correct entry. This results in a previously unknown information leakage in this application. A malicious webpage, whose title matches the title of an entry present in the database, will be able to obtain the corresponding username/password information. We tested this mechanism by creating a dummy webpage with the same title as a secure entry, and we were able to transfer the corresponding login information to the dummy webpage.

6.8.4.2 thttpd

thttpd is a small web-server application. Previous dynamic information flow tracking methods have demonstrated the leakage of password information due to *thttpd*'s inherent authentication mechanism [147]. Here, we demonstrate the presence of this leakage using DemandFlow.

thttpd stores the authentication information in an internal database. Any connection request is first validated by comparing the username and password specified by the user with the internal database. We assign the global variable corresponding to the database file with the *confidential* label. The network procedures used by *httpd* for connecting to the user (e.g. *send_authenticate*) are marked as *untrusted* program locations. The mapping of the symbolic lattice onto the lattice described above reveals that arguments to *untrusted* procedures, both at program locations where the authentication is valid and invalid, contain *confidential* labels.

The comparison of the above result with the previous method [147] highlights an inherent limitation of static information flow systems over dynamic systems.

App	Lang	LOC	# Proc	Time (s)	Mem (MB)	FP Fmt Str	FP Dir Trav
bwaves	F	715	22	0.1	.15	0	0
lbm	C	939	30	0.1	0.17	0	0
mcf	C	1695	36	0.1	0.2	0	0
libq	C	2743	73	0.8	0.5	0	0
leslie3d	F	3024	32	1.0	2.5	0	0
namd	C++	4077	193	0.5	0.50	0	1
astar	C++	4377	111	1.5	0.25	0	0
bzip2	C	5896	51	1.2	1	0	0
milc	C	9784	172	1.2	1.56	0	0
sjeng	C	10628	121	1.4	2.6	0	2
sphinx	C	13683	210	1.3	3.5	0	0
zeusmp	F	19068	68	3.1	3.75	0	0
omnetpp	C++	20393	3980	22.9	87.5	0	0
hmmr	C	20973	242	1.5	3.72	0	0
soplex	C++	28592	1523	1.5	10.3	0	0
h264	C	36495	462	4.2	26.4	0	0
cactus	C	60452	962	3.1	8.3	0	1
gromacs	C/F	65182	674	9.6	46.1	1	0
deallI	C++	96382	15619	1.2	3.1	0	0
calculix	C/F	105683	771	20.1	70.7	0	0
tonto	F	108330	4086	3.5	5.8	0	0
povray	C++	108339	3678	1.8	3.1	0	0
perlbench	C	126367	2183	24.6	59.7	0	0
gobmk	C	157883	4188	12.9	30.2	0	2
gcc	C	236269	6426	62.4	110.5	0	0
xalan	C++	267318	14441	4.5	9.82	0	0

Figure 6.11: *Spec Benchmarks*

RIFLE [147], being a dynamic method, assigns an individual label to each different username and password. The experiments, as presented by Vachharajani et al. [147], demonstrate that in the case of an unauthorized user access, the reply consists of the labels of all the usernames since the whole file is scanned. In the case of an authorized access, the reply contains the labels of usernames upto the authorized username in the database and the password of the current user.

On the other hand, a static framework like DemandFlow does not have any access to data and can only track the leakage of statically visible information in a program. It is not possible to assign the labels corresponding to individual elements in the database, the label can be applied at the granularity of the complete database. Consequently, DemandFlow establishes a somewhat coarser information flow leakage

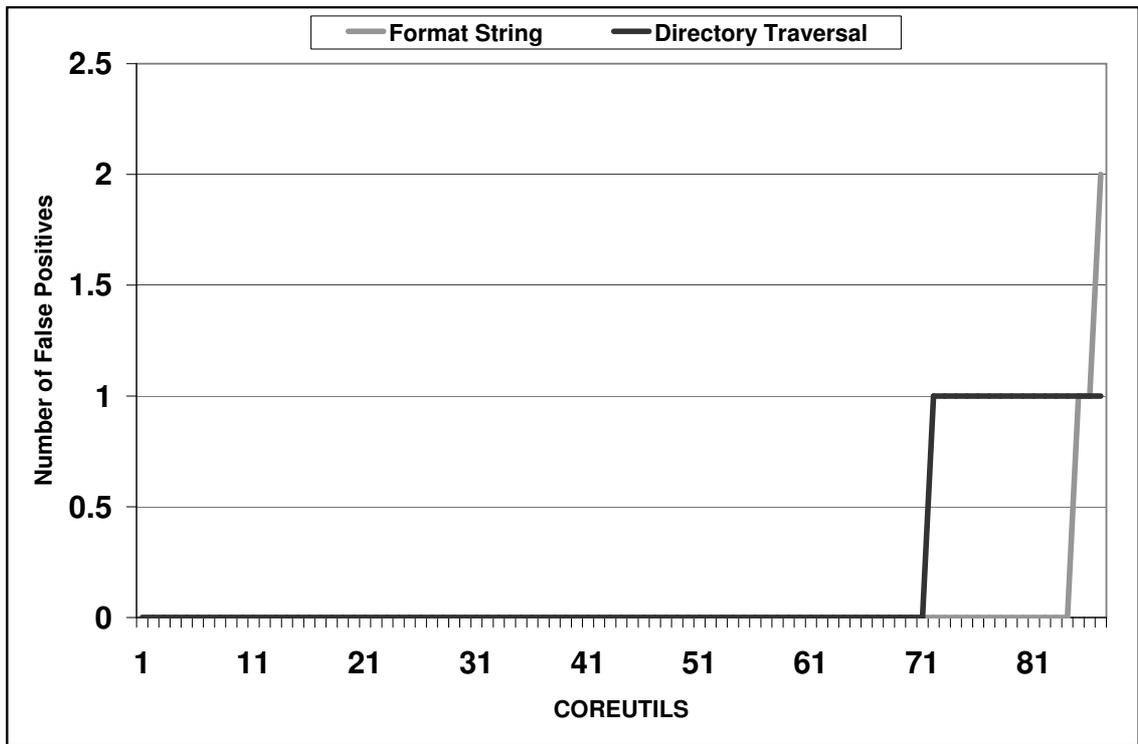


Figure 6.12: *Vulnerability detection in Coreutils*

as compared to RIFLE.

6.8.5 Spec Benchmarks and Coreutils

In this section, we demonstrate DemandFlow’s scalability by applying the analysis on the complete SPEC benchmark suite. SPEC benchmarks suite contain several large real-world applications and comprise a diverse set of real and environment intensive applications.

Fig 6.11 lists the running time, storage requirements and possible vulnerabilities reported by DemandFlow for each of the programs in SPEC benchmark suite. As evident from the table, the analysis time is typically low, under a minutes all the benchmarks. The storage requirements for most of the benchmarks are under

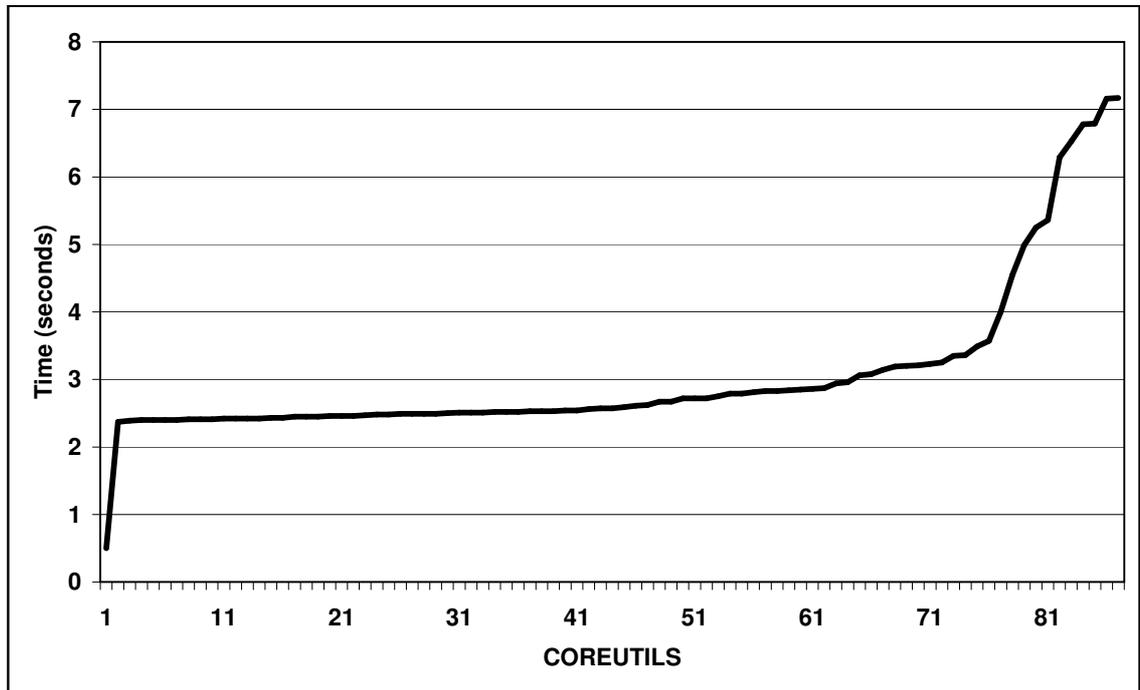


Figure 6.13: *Time for analyzing Coreutils*

100 MB, well within the memory available on modern systems. Similar to *KeePassX* example, the time and storage requirements are typically lower for C++ and Fortran programs due to relatively smaller number of *format string* calls. A limited overall storage requirement validates the computation of information flow expressions instead of a single-bit taint information, as employed by most information-flow frameworks. DemandFlow raises only one false alarm regarding format-string vulnerability and only a few misleading instances of directory traversal vulnerability for the complete set of the benchmarks. This translates to an extremely low false-positive rate of .007 per 1000 lines of code ($7/10^6$).

The statistics presented in Fig 6.12 demonstrate that DemandFlow reports three instances of format-string vulnerability in complete Coreutils suite. One on these three instances is a possible true positive (`csplit` in Fig 6.5). Further, it only

reports 15 false samples for directory traversal vulnerability. This translates to a low false-alarm rate of 0.107 per 1000 lines of code (18/140,000). and all 89 programs in the latest stable version of GNU Coreutils (Version 8.10) while Coreutils and form the core user-level environment installed on many Unix systems.

Fig 6.13 plots the running time for analyzing GNU Coreutils. The time spent ranges from 0.5 seconds for smaller applications like *test* to 7 seconds for larger ones like *shasum*.

Chapter 7: Cache Locking

7.1 Introduction

Modern embedded systems employ several memory technologies to meet stringent run-time and power consumption constraints. SRAM and DRAM are the two most common memories used for storing program code and data. Due to the relative cost and performance of these memories, a large amount of DRAM is often complemented with a small-size on-chip SRAM. The proper use of SRAM in embedded systems is imperative in meeting run-time and energy constraints.

SRAM is most commonly managed in the form of a hardware-cache. A cache dynamically stores a subset of the frequently used data or instructions following a fixed replacement policy.

Various different approaches have been suggested to enable software involvement in the management of on-chip memory. One approach involves the addition of lightweight software-controlled memory like Scratchpad memory (SPM) which rely on explicit compiler support for data allocation. Another approach involves explicit modifications to the cache memory structure and availability of programmer level cache control instructions to enable direct software involvement in cache

replacement decisions.

On similar lines, several embedded systems like Intel's XScale and ARM's latest cortex processors provide the facility of locking one or more lines in the cache - this feature is called *cache locking*. An address, once locked in the cache, always results in a hit on subsequent accesses unless an unlocking operation is explicitly carried out. Thus, the software can influence the replacement decision made by the cache and thereby alleviate the potential mistakes resulting from cache hardware management. As an example, suppose a soon-to-be-accessed element is susceptible to replacement according to the underlying cache replacement policy in favor of an element that will not be accessed soon, locking this element in the cache will result in a better cache performance.

However, current methods regarding instruction cache locking are geared towards improving real-time predictability of applications [117, 116, 68, 151]. These methods employ instruction cache locking for adapting the cache to multi-task real time systems.

We presented the first method in literature [9] employing instruction cache locking as a mechanism for improving the average-case run-time of general embedded applications, thus widening its applicability beyond hard real time systems. Our scheme is implemented inside a binary rewriter; hence is applicable to binaries compiled using any compiler or software development toolchains and to programs whose source code is not available e.g. legacy code or third party software. Cache locking technique can be applied to both instruction and data caches but in this work, we limit ourselves to the problem of instruction cache locking.

Liang and Mitra [98] extended our earlier work [9] and presented an optimal algorithm for static instruction cache locking. However, both these methods only explore static cache locking, where instructions are locked once before the start of the program and remain locked during the entire execution of the program.

In this work, we extend our earlier instruction cache locking mechanism and propose a novel dynamic cache locking algorithm, where the addresses locked in the cache are updated dynamically during the execution of a program. Our mechanism identifies the program points with significant shift in program locality and employs a cost-driven model to compute the set of lines which should be locked at each such program point. The input program is instrumented to achieve the locking of required lines at each program point. This mechanism accounts for changing program requirements at runtime and dynamically modifies the cache content.

We also demonstrate that an optimal solution to dynamic instruction cache locking can be obtained in polynomial time, contrary to the previously held belief [9] about instruction cache locking being a NP-complete problem. However, as we will discuss in later sections, such an algorithm cannot be implemented practically with current support for instruction cache locking. Hence, we propose a heuristic based approach for deriving a solution for dynamic cache locking.

The rest of the section is organized as follows. Section 7.2 describes the underlying cache locking interface. Section 7.3 overviews related work and lists the advantages of our method. Section 7.4 presents a small example to depict the benefit of instruction cache locking. Section 7.5 formalizes the cache locking problem and its complexity. Section 7.6 presents our solution for static cache locking while Sec-

tion 7.7 presents the dynamic counterpart. Section 7.8 presents an overview of our implementation framework. Section 7.9 presents our method’s results for different cache and architecture configurations on a variety of benchmarks.

7.2 Cache Locking Interface

There are two most common kind of locking mechanisms present in modern embedded systems - way locking and line locking. *Way locking* is a coarse grain approach to cache locking where locking is available at the granularity of ways of a set-associative cache. Locking a particular way in cache implies the way is locked in each set of the set-associative cache. This kind of locking is present in ARM’s cortex processors [15] and ARM11 family of processors [14].

Line locking is a more fine-grained approach to cache locking. In this interface, the locking mechanism is available at the granularity of single cache line as opposed to single way in way locking. In this interface, it is possible to have a different number of locked lines in different sets of the cache. Intel’s XScale [159], ARM9 family and BlackFin 5xx family processors [30] support this kind of locking mechanism.

In this work, we explore the line locking interface present on embedded systems. These platforms provide special co-processor-based lock instructions for locking an address specified as their argument in the cache. In such processors, way 0 of the cache can’t be locked; we respect this constraint in deriving our results. However, we emphasize that our method does not require any such constraint and can be applied for locking lines in all the ways of any set.

7.3 Related Work

There are many existing methods targeting improvement of on-chip memory performance through software involvement. Research in this direction can be broadly categorized in two approaches: (i) approaches involving an additional software-controlled memory apart from, or instead of, the cache; and (ii) approaches involving direct modifications of the cache memory structure.

The first category of methods involve modifications to the memory hierarchy by introducing additional software-controlled memories like Scratchpad memory (SPM) and loop caches. Various different kind of methods have been suggested for managing the data to be placed in SPM [134, 24, 112, 152, 153, 138, 18, 146]. A *loop cache* [70] is a small instruction buffer which can be pre-loaded with frequently executed loops and functions thus accelerating their access-time during program execution. SPMs and loop caches are used in industry primarily where the run-time behavior of applications is predictable; or to improve real-time performance. Caches are better at tracking run-time behavior; hence are widely used in many non-real-time and soft-real-time systems.

Even though cache locking tries to achieve the same goal of improving local memory performance, its management strategy is inherently different from the allocation problems for the above software-controlled memories such as SPM. There are two reasons for that. First, when a cache locking method decides to lock a line in the cache, other lines that conflict with it can no longer reside in cache in a direct-mapped cache, or have reduced number of slots available in a set-associative

cache. This opportunity cost does not occur, and is not modeled, by methods for SPM allocation. In contrast, our cache locking method inherently models this cost. Correctly modeling this opportunity cost is crucial – a SPM allocator oblivious to this cost when used for locking could exclude heavily used lines from cache, leading to poor run-time. Another reason that SPM allocators are not suitable for cache locking is that a particular element can be placed at any location in SPM, whereas the cache hardware decides the location of each element in a cache. This results in entirely different kinds of constraints for the cache locking problem. The energy model in terms of cache hits and misses suggested in [152] for cache-aware SPM allocation is somewhat similar to the time model we present but their method addresses a completely different problem.

In the second category, there are methods that involve modifications to the cache hardware to equip software to dynamically modify cache replacement decisions. Rudolph et al [44] introduce column caching, to provide software the ability to dynamically partition the on-chip memory into scratchpad memory; Wang et al [126] proposed the extension of each cache line with evict-me and kill-me bits; along with a compile time locality analyzer to determine their values. These methods provide interesting ideas for improving cache performance but rely on hardware modifications that are unavailable in any commercial processors. In contrast, our method is a software-only scheme applicable to a variety of commercial processors.

Research has been carried out to exploit the cache features present in existing hardwares - locking is one such kind of feature available in modern embedded systems. Hollander et al [29] suggested reuse-distance-based methods for generating

cache hints for memory access instructions, available in EPIC architectures, resulting in improved data cache performance. In contrast, we don't target the hardware with cache hints; rather we target cache locking hardware.

Instruction cache locking has primarily been employed as a mechanism for adapting the cache to multi-task real time systems. In multi-task systems, the presence of caches leads to unpredictability and results in extreme over-estimation of worst case execution time, as each access can result in a miss in the worst case [117]. I-cache locking has been employed in such scenarios to provide predictability; thus improving the worst case estimation. The objective of the cache-content selection problem in such scenarios is to improve the worst case system behavior according to some of real-time schedulability metrics as described in [117, 116, 68, 151, 16, 40]. In contrast, our objective of cache-content selection is to improve average case runtime of embedded applications which is completely different objective, requiring a very different strategy.

There has been very little research on using cache locking for performance improvement of general embedded applications. Hu et al [161] presented a method for data cache locking in Itanium and XScale processors based on the length of the reference window for each data-access instruction. In contrast, we present a locking scheme for the instruction cache. Further, their method doesn't involve finding the optimal number of cache lines to be locked in the cache; rather they rely on locking every possible line which can be locked in cache. The over-aggressive locking might provide negative results and does not ensure that the locked cache would give perform better than cache with no locking. Our method suitably addresses these

limitations.

Earlier, we had presented the first method [9] in literature employing instruction cache locking as a mechanism for improving the average-case run-time of general embedded applications. However, our previous work only explored a static solution to cache locking. Liang and Mitra [98] extended our work and presented an optimal strategy for static instruction cache locking. Later, Liu et al [100] also present a method for employing instruction cache locking for improving average case performance. However, their model does not model cache conflicts and is only applicable to fully associative caches. Their method relies on techniques like code positions to eliminate the conflicts. In contrast, our method directly models conflicts and is applicable to a cache with any associativity.

In this work, we extend our previous work [9] and propose a dynamic solution for cache locking, which accounts for changing program requirements at runtime and updates the instructions locked in cache dynamically with program execution.

We summarize the benefits of our scheme: (i) ours is the first method for employing instruction-cache locking as a mechanism for improving the average case run-time of general embedded applications, thus widening its applicability beyond hard real time systems. (ii) ours is the first dynamic method for instruction cache locking, enabling better results than static schemes. (iii) we provide a profile-based method and derive the cost-benefit from actual cache statistics; thus our method is guaranteed to improve over the performance of cache without locking. (iv) our method has been implemented inside a binary rewriter, widening its applicability to binaries compiled using any compiler. (v) our method has an inherent mechanism

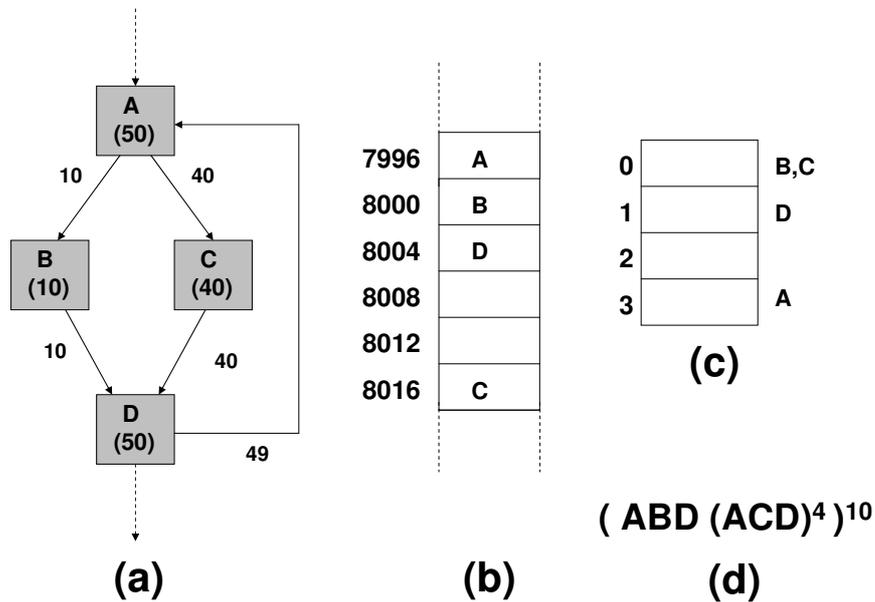


Figure 7.1: (a) *Weighted CFG of a small part of a program. A, B, C and D are instructions of 4 byte each* (b) *A hypothetical memory layout of the above instructions* (c) *A dummy 16-byte direct mapped instruction cache. The alphabets at right hand side of each cache line show the instructions which are mapped to the line according to the cache mapping function* (d) *The execution trace of this part of the program*

that determines the optimal number of cache lines to be locked - it does not lock each possible cache line, as suggested by some previous methods. (vi) cache locking is already available on existing hardwares and thus our method does not entail any new hardware modifications, making our approach readily applicable.

7.4 Motivation

In this section, we present the potential benefits of instruction-cache locking in improving cache efficiency via a small example. Figure 7.1 shows a weighted control-flow graph (7.1(a)) and execution trace (7.1(d)) of a small part of a program; its

hypothetical memory layout (7.1(b)) and a dummy cache configuration (7.1(c)). The nodes and edges of the control-flow graph are labeled with their execution frequencies as observed during a profile run of the program. The execution trace (7.1(d)) of the program reveals that a single execution of node B is followed by four instances of node C. This sequence of execution of node B followed by node C is repeated 10 times during the execution of the program. For simplicity, we assume that nodes A, B, C and D contain only a single instruction each. For ease of explanation, the instruction cache is a tiny 16-byte direct mapped cache with one word per line. The addresses are mapped to the cache lines according to the standard modulo-based cache mapping function:

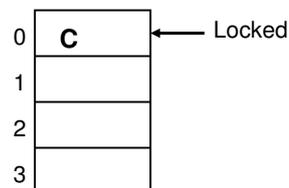
$$Set = (addr) \bmod \frac{Cache-Size}{Associativity * Words-Per-Line} \quad (7.1)$$

According to the above cache mapping function and the memory layout, instructions B and C share the same line in the cache. During the execution of the above program, node B and node C alternately keep replacing each other in the cache, resulting in a large number of cache misses. The second column in Figure 7.2(a) shows that this cache configuration leads to 22 misses for this sample program.

Next, assume the presence of locking functionality inside the instruction cache. If node C is locked into cache line 0 then C would not be replaced by node B during the execution of the program. Node C would observe only one compulsory miss while number of misses for B would remain the same. The third column in Figure 7.2(a)

Node	Number of Misses without locking	Number of Misses with locking
A	1	1
B	10	10
C	10	1
D	1	1
Total	22	13

(a)



(b)

Figure 7.2: (a) Number of misses observed for each node with and without locking
 (b) Locking of node C in set 0 of cache

shows the number of misses observed by each node when node C is locked in cache as shown in Figure 7.2(b). With cache locking, we observe only 13 misses, down from 22 misses in cache without locking. This example highlights the potential of I-cache locking as an effective mechanism for reducing cache misses.

7.5 Theoretical Analysis of Cache Locking

The cache-locking problem involves selecting the memory addresses which should be locked in the cache during each time interval, and the program locations for locking and unlocking, such that the total number of instruction cache misses over the lifetime of the program is minimized. The solution to this problem is influenced by the behavior of the cache mapping function. In a set-associative cache, an address is mapped to the cache line according to the cache mapping function (7.1). For a given memory address, this function returns the cache set where the address is mapped in

the cache. A particular memory address always gets mapped to the same set in the cache, given by the above function. Thus, given the full range of instruction-memory addresses in the current program, the list of addresses which get mapped to a set during the lifetime of the program can be accurately obtained for each cache set. Once this mapping of addresses to the corresponding set is obtained, *each cache set can be independently analyzed to determine the memory addresses to be locked in that set.*

At each time instant T , the cache locking problem has two objectives (i) determining the number of lines, L , which should be locked in this set (ii) selecting L virtual cache lines out of total candidates which should be locked in the set.

In his seminal paper [27], Belady proposed an optimal offline replacement policy for virtual memory pages, which has been subsequently widely applied for cache analysis as well. Belady's algorithm achieves the lowest possible cache miss rate. Other faster algorithms [140] have also been proposed to achieve the optimal cache miss rate. Collectively, the class of such algorithms is referred to as **OPT** algorithm. We demonstrate that the **OPT** algorithm can be employed to provide an optimal solution for cache locking problem in each cache set.

The **OPT** algorithm analyzes the cache accesses in the trace in the execution order. The resulting replacement decisions can either be employed for improving cache performance in future executions or for comparing different cache strategies. Intuitively, given a trace of block accesses for the program, the **OPT** algorithm is based on evicting the block which will be referenced furthest in the future. Consider an address X that is referenced twice in a program trace at times t_1 and t_2 , $t_2 \geq t_1$.

According to OPT algorithm, the decision for keeping X in the cache during the time interval $\{t_1, t_2\}$ is taken at t_2 (and implemented at t_1 in future executions). X is not kept in the cache if the total number of elements already in the cache in time interval $\{t_1, t_2\}$ is equal to the cache capacity; otherwise it is kept in this interval.

We observe that the ability to lock an address in the cache provides a tangible mechanism to implement the solution proposed by OPT algorithm. For example, in order to keep an element X in the cache during the time interval $\{t_1, t_2\}$, X can be locked in the cache at t_1 and unlocked at t_2 . Consequently, cache locking mechanism actuates the implementation of OPT algorithm. Based on this observation, we state the following important lemma.

LEMMA: An optimal solution for cache locking can be derived in a polynomial time, assuming perfect prior knowledge of memory accesses.

PROOF: OPT is a polynomial time algorithm for obtaining an optimal solution for cache performance. In other words, the OPT algorithm minimizes the number of misses in the cache. The Cache locking problem shares the same goal of minimizing the number of misses in the cache. The capability of locking a line in cache enables the implementation of each step of OPT algorithm in a constant time. Hence, OPT is an optimal solution for the cache locking problem as well. The polynomial time complexity of OPT results in a polynomial-time optimal algorithm for cache locking.

A perfect (or complete) knowledge of future memory accesses enables OPT algorithm to make optimal replacement decisions. Extending this optimal replacement

algorithm to the cache locking problem implicitly models the opportunity cost arising due to precluding the remaining elements from the cache during cache locking since OPT considers *every* cache line as a possible candidate for locking.

However, the mechanism of inserting a cache locking instruction for locking an element in the cache, as presented in Section 7.2, generates several pragmatic challenges. The OPT algorithm provides a set of addresses which should be locked in the cache at each program instant. A direct instrumentation of the program with the instructions for locking these addresses changes the program layout, invalidating the results provided by OPT algorithm. The other option is to leave a placeholder before each instruction in the program. These placeholders can later be employed for inserting cache locking instructions as per the results of OPT algorithm. The remaining placeholders can be replaced by a NOP instruction. However, the presence of large number of such NOP instructions results in a huge overhead in execution-time, negating the improvement in memory performance due to cache locking.

Consequently, the cache locking mechanism present in current hardware leaves us in a conundrum where the optimal algorithm cannot be implemented. Hence, we propose two distinct solutions to overcome this practical challenge:

→ **Static Cache Locking:** We formulate a static solution to instruction cache locking where instructions are locked once before the start of the program and remain locked during the entire execution of the program. This solution obviates any requirement of changing the program layout.

→ **Dynamic Cache Locking:** In this formulation, we obtain an hybrid be-

tween static locking and OPT algorithm. Instead of inserting placeholders at each point in the program, we insert placeholders only at judiciously chosen program points. Such program points are chosen based on a possibility of a large change in program locality. These placeholders are later replaced by cache locking instructions or NOP as per the requirement.

Section 7.6 presents the solution for Static Cache Locking. Section 7.7 extends this solution to obtain Dynamic Cache Locking.

7.6 Static Cache Locking

In this section, we formalize the cache locking problem as an optimization problem and explain our cache locking algorithm in detail. We present a static solution to instruction cache locking where instructions are locked once before the start of the program and remain locked during the entire execution of the program.

Since elements in the cache are locked at the granularity of cache lines and not individual memory addresses, addresses need to be analyzed in terms of cache lines. In order to mathematically represent this situation, we introduce a new concept of virtual cache line. Given an instruction address, *addr*, the **Virtual Cache Line** is defined as

$$\text{Virtual Cache Line} = \frac{\textit{addr}}{\textit{Words-Per-Line}} \quad (7.2)$$

The remaining analysis for cache locking is carried out in terms of virtual cache lines. We introduce the following definitions to ease the explanation

N : Associativity of the cache; \mathbf{s} : A cache set

$X_{\mathbf{s}}$: Set of virtual cache lines which get mapped to set \mathbf{s}

M : Cardinality of set $X_{\mathbf{s}}$.

K : Hardware specified limits on maximum number of lines which can be locked in a set

L : Number of lines to be locked, $L \leq K$

The static cache locking problem has two objectives (i) determining L : the number of lines which should be locked in this set (ii) selecting L virtual cache lines, out of M candidates, which should be locked in the set.

If L lines are locked in this set, L locked virtual cache lines result in L compulsory misses and no other misses are observed for these lines. The remaining $M - L$ virtual cache lines from set $X_{\mathbf{s}}$ perceive the cache as a $N - L$ set associative cache. In case the total number of virtual cache lines sharing this particular cache set is more than the associativity of the cache, this decreased associativity might result in an increased miss rate for the remaining lines.

The number of solutions to the cache-locking problem is exponential since there are an exponential number of ways to choose up to K lines to lock out of M contenders. In all likelihood, this is a classical NP Hard combinatorial optimization problem, which does not have an exact solution, although we have not attempted to formally prove this. Further, finding an exact solution is complicated by the

fact that the increased miss rate for remaining $M - L$ virtual cache lines cannot be accurately determined unless we know which virtual cache lines are locked in the current set of the cache, which is one of the objectives of this optimization problem. Hence, an exact solution will not only have an exponential number of solutions, but will require a profiling run for each solution to determine the increased miss rate for the remaining unlocked lines, which is completely infeasible. Consequently, we explore an approximate solution for this problem, as presented below.

7.6.1 Cache Locking Algorithm

Here, the solution for one cache set is considered in detail; the same method is employed repeatedly for each set. Our solution is based upon the total time taken to access each virtual cache line during the lifetime of the program. We introduce a time model for representing the total time taken to access a particular virtual cache line during the lifetime of the program in presence of locking.

LOCKLIST: The running list of virtual cache lines locked so far in the set.

LL: The number of elements in list **LOCKLIST**.

HIT_{LL}(x_i)/MISS_{LL}(x_i): Total number of hits/miss obtained for a virtual cache line x_i assuming that LL number of lines were locked in the current set

F(x_i): The total number of accesses to a virtual cache line x_i .

T_{HIT}/T_{MISS}: Hit and Miss latency of the cache, respectively, in processor cycles.

Mathematically, this model is described as

$$\begin{aligned} Time(x_i|LOCKLIST) = & HIT_{LL}(x_i) * T_{HIT} \\ & + MISS_{LL}(x_i) * T_{MISS} \end{aligned} \quad (7.3)$$

In our notation, $\mathbf{Time}(A|B)$ is the total time to access virtual cache line **A** during the lifetime of the program given that all the virtual cache lines in mathematical set **B** have already been locked in **A**'s cache set. (This notation is borrowed from conditional probability.). **LOCKLIST** is initialized as an empty list. Every time a line is selected to be locked, the **LOCKLIST** is updated with the line. The analysis presented is only applied to $\mathbf{x}_i \notin \mathbf{LOCKLIST}$.

In order to find the virtual cache lines which should be locked in this set, we introduce a cost-benefit model based on the above time model to find the net benefit (benefit - cost) of locking a particular cache line. The following relation between number of accesses, number of hits and number of misses always holds true, irrespective of the number of lines currently locked (**LL**) in the set:

$$F(x_i) = HIT_{LL}(x_i) + MISS_{LL}(x_i) \quad \forall LL \quad (7.4)$$

Using the above relation and the time model from equation (7.3), the original access time for virtual cache line \mathbf{x}_i , assuming that virtual cache lines in **LOCKLIST**

are already locked in this set, can be represented as:

$$\begin{aligned} Time(x_i|LOCKLIST) &= HIT_{LL}(x_i) * T_{HIT} + \\ &(F(x_i) - HIT_{LL}(x_i)) * T_{MISS} \end{aligned} \quad (7.5)$$

If line x_i is locked in cache, only one miss (a compulsory miss) would be observed for this line. All the remaining accesses to this line would definitely result in a hit. Thus the new access time for this line would be given by following relation:

$$\begin{aligned} Time(x_i|(LOCKLIST \cup \{x_i\})) &= T_{MISS} + \\ &(F(x_i) - 1) * T_{HIT} \end{aligned} \quad (7.6)$$

Subtracting equation (7.6) from equation (7.5), the potential benefit of locking a particular line x_i can be expressed as:

$$\begin{aligned} BenLock(x_i) &= Time(x_i|LOCKLIST) \\ &- Time(x_i|(LOCKLIST \cup \{x_i\})) \\ &= (F(x_i) - HIT_{LL}(x_i) - 1) \\ &* (T_{MISS} - T_{HIT}) \end{aligned} \quad (7.7)$$

In order to calculate the cost of locking a line, we only consider the opportunity

cost of locking a line and not the actual cost of executing locking. Since we are just considering a static solution, the cost of executing a single locking instruction is negligible and hence does not affect our analysis.

In order to represent the opportunity cost of locking a particular cache line, we need to model the increase in total access time for the remaining virtual cache lines which map to the set under consideration. So far, $|\text{LOCKLIST}| = \text{LL}$ virtual cache lines have been selected for locking. Let, \mathbf{X}_{s_i} denotes the set of virtual cache lines mapped to the current cache set s_i , excluding the LL elements in the list LOCKLIST . The elements in LOCKLIST are already locked in cache, hence they won't observe any opportunity cost.

According to above terminology, each line $x_j \in \mathbf{X}_{s_i}$ observes $\text{HIT}_{\text{LL}}(x_j)$ hits. Each element belonging to set \mathbf{X}_{s_i} is a potential candidate for locking. If line x_i is locked at this step, then each remaining element x_j of set \mathbf{X}_{s_i} would observe a lesser number of hits, denoted by $\text{HIT}_{\text{LL}+1}(x_j)$. This constitutes the cost of locking a particular line x_i . Mathematically, for each $x_j \in \mathbf{X}_{s_i}$, the original access time is represented by equation (7.5). The new access time after locking line x_i can be represented as:

$$\begin{aligned} \text{Time}(x_j | (\text{LOCKLIST} \cup \{x_i\})) &= \text{HIT}_{\text{LL}+1}(x_j) * T_{\text{HIT}} + \\ & (F(x_j) - \text{HIT}_{\text{LL}+1}(x_j)) * T_{\text{MISS}} \end{aligned} \tag{7.8}$$

The increase in access time for one element x_j due to locking the line x_i ,

denoted by $\mathbf{Cost}_{\text{Lock}(x_i)}(x_j)$, can be represented as

$$\begin{aligned}
\mathbf{Cost}_{\text{Lock}(x_i)}(x_j) &= \mathit{Time}(x_j | (\mathit{LOCKLIST} \cup \{x_i\})) \\
&- (\mathit{Time}(x_j | \mathit{LOCKLIST})) \\
&= (\mathit{HIT}_{LL}(x_j) - \mathit{HIT}_{LL+1}(x_j)) \\
&* (T_{\text{MISS}} - T_{\text{HIT}})
\end{aligned} \tag{7.9}$$

The total cost of locking the line x_i can be represented as

$$\mathbf{CostLock}(x_i) = \sum_{(x_j | x_j \in X_{s_i} \& x_j \neq x_i)} \mathbf{Cost}_{\text{Lock}(x_i)}(x_j) \tag{7.10}$$

The net benefit of locking a particular virtual cache line can be calculated as

$$\mathbf{NetBenefit}(x_i) = \mathbf{BenLock}(x_i) - \mathbf{CostLock}(x_i) \tag{7.11}$$

A positive **NetBenefit** for a cache line implies that locking this line would result in a lesser total memory access time for the program. Magnitude of the **NetBenefit** represents the change in total access time. Thus the cache line with maximum positive benefit is the ideal candidate for locking at this step.

In order to meet the both the objectives of the problem – determining the

N : Cache size in number of lines
 K : Number of lines which can be locked in each set
 S : Number of sets in the cache.
 s_i : Set where memory address x_i gets mapped
 X_{s_i} : The set of memory addresses which get mapped to set s_i
 $F(x_i)$: Total number of memory accesses to address x_i
 LL : Iterator over number of lines locked in one set
 $HIT_{LL}(x_i)$: Total number of hits obtained for x_i when LL lines are locked in set s_i
 $MISS_{LL}(x_i)$: Total number of miss obtained for x_i when LL lines are locked in set s_i
 $LockList(s_i)$: Set of virtual cache lines which should be locked in set s_i
 $NumLockLines(s_i)$: Number of virtual cache lines which should be locked in set s_i .
 T_{HIT} / T_{MISS} : Hit/Miss latency in processor cycles

```

void Cache_Locking_Algorithm() {
1.  for(each set  $s_i$  in range 0 to S -1 ) do {
2.      for(each LL in range 0 to K -1 ) do {
3.          for (each  $x_i$  in  $X_{s_i}$ ) {
4.               $BenLock(x_i) = (F(x_i) - HIT_{LL}(x_i) - 1) * (T_{MISS} - T_{HIT})$ 
5.               $CostLock(x_i) = \sum_{x_j|x_j \in X_{s_i} \& x_j \neq x_i} CostLock(x_i)(x_j)$ 
6.               $NetBenefit(x_i) = BenLock(x_i) - CostLock(x_i)$ 
7.          }
8.          If there exists a  $x_k$  such that  $NetBenefit(x_k)$  is maximum and is positive.{
9.              Add  $x_k$  to  $LockList(s_i)$ 
10.              $NumLockLines(s_i) = NumLockLines(s_i) + 1$ 
11.              $X_{s_i} = X_{s_i} - x_k$ 
12.          }
13.          else {
14.              break; //Locking done for this set
15.          }
16.      }
17. }
18.return
19.}
  
```

Figure 7.3: *Static Cache Locking Algorithm.*

number of cache lines to be locked in the set and selecting the virtual cache lines to be locked in these lines of the set – we devise a greedy and iterative solution for this problem. Let us examine the steps taken at the $(LL + 1)^{th}$ iteration. At

this point, we have a list `LOCKLIST` of LL virtual cache lines which should be locked in the set. The above model is used to calculate the `NetBenefit` for each of the virtual cache line $x_i | x_i \in X_{s_i}$. If the net-benefit is negative for all the elements, the locking is discontinued for this set, implying that it is not beneficial to lock any more cache line in this set. The running list `LOCKLIST` represents the final list of virtual cache lines which should be locked in this set. If there is at least one element with positive net-benefit, we find the virtual cache line which has maximum net benefit for locking. This line is added to the list `LOCKLIST` and is removed from the locking candidates set X_{s_i} . The above steps are repeated at each iteration until at least one of the following two conditions is true: (i) we reach the limit of maximum cache lines which can be locked in a set or (ii) we reach a point where the net benefit becomes zero for each virtual cache line in this set. At the end of this process, we get the number of cache lines (`|LOCKLIST|`) as well as memory addresses which should be locked in this set (`LOCKLIST`). In other words, we obtain the solution for both the unknowns of cache locking problem. Figure 7.3 describes the psuedocode for the cache locking algorithm.

In the above cost-benefit model, HIT_{LL+1} cannot be determined precisely till we know which virtual cache line gets locked during the current step of iteration and would be different for each virtual cache line. Determining the exact value is completely infeasible given that the number of profile runs needed would equal the number of virtual cache lines, which is a very large number. Thus, an approximate value of HIT_{LL+1} is obtained by locking a dummy (unused) virtual cache line in the set apart from LL lines already locked. Nevertheless, this approximation always

provides conservative estimates for future hit rate – in reality, one less virtual cache line would be competing for space in cache – and thus locking a line is guaranteed to show performance improvement.

7.7 Dynamic Cache Locking

In this section, we discuss our mechanism for dynamically updating the contents locked in a cache. As mentioned in Section 7.5, our dynamic solution is based on possibly changing the locked contents of cache at the program points having a possibility of a significant change in the program locality.

The solution consists of the following steps. First, the program code is analyzed to determine a set of promising program points. Second, a timestamp is associated with every program point such that the program points are reached during runtime in the timestamp order. Third, the timestamps are updated to discard the program points with high execution frequency since they will result in high locking overheads. The code between two consecutive timestamped program points represents a code region. Regions correspond to the granularity at which cache locking decisions are made. Next, a heuristic based algorithm is employed to compute the locked content in the cache at these refined program points. The cache content is fixed in a particular region, but may change at region boundaries. We first describe our method for determining such program points (and regions) and then propose our solution to determine the locked cache content in each particular region.

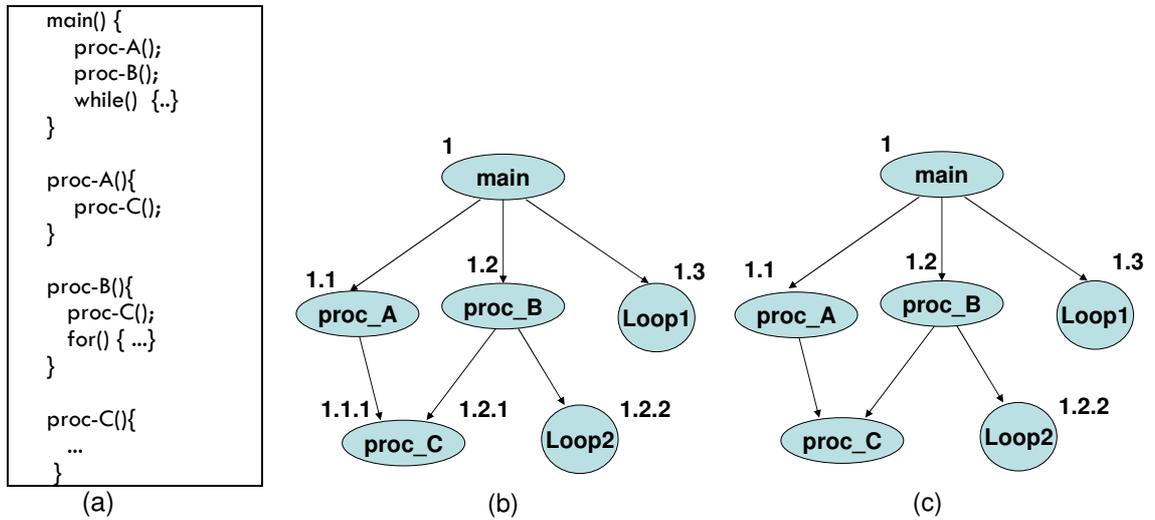


Figure 7.4: Example showing (a) a program outline; and (b) its DPRG showing nodes, edges & timestamps (c) modified DPRG nodes and timestamps assuming that execution frequency of *proc-C* is greater than *LIMIT*

7.7.1 Program Points

The choice of program points is critical to the success of the algorithm. Promising program points are those after which the program has a significant change in locality behavior. Further, the dynamic frequency of program points should be less than the frequency of regions, so that the cost of executing cache locking instructions can be recouped by corresponding improvement in memory performance. Hence, sites just before the start of loops are especially promising program points since they are infrequently executed compared to the insides of loops. Moreover, the loop often re-uses instructions, justifying the cost of locking lines in the cache.

With the above considerations, we employ a modified version of Data-Program Region Graph (DPRG), proposed by Udayakumaran and Barua [145], for determining program points. We modify the original DPRG structure [145] in two aspects.

First, the original DPRG was proposed to solve the data allocation problem, hence it also represents variable accesses. We do not need to represent variables since we are only targeting instruction cache locking. Second, we refine the program points obtained through original DPRG structure by discarding the program points with high execution frequency, since locking at those locations results in high overheads. The threshold for refining the program points is heuristically determined using profiling, as explained in later sections. Below, we summarize the DPRG structure and our modifications to this structure.

DPRG defines program points as (i) the start and end of each procedure; (ii) just before and just after each loop (even inner loops of nested loops). In this way, program points track most major control-flow constructs in a program. Program points in a DPRG are the only initial candidate sites for applying cache locking in the ensuing region. This set is further refined and the actual solution regarding the elements to be locked in each region is governed by the method proposed in Section 7.7.2.

Figure 7.4 shows an example illustrating how a program is divided into regions and then marked with timestamps. Figure 7.4(a) shows the outline of an example program. It consists of four procedures, namely `main()`, `proc-A()`, `proc-B()` and `proc-C()` and two loops, `Loop1` and `Loop2`.

Figure 7.4(b) shows the Data Program Relationship Graph (DPRG) (excluding variable accesses) for the program in figure 7.4(a). The DPRG data structure helps in the marking of timestamps and the identification of regions. The DPRG is essentially the programs call graph appended with new nodes for loops. In the

DPRG shown in figure 7.4(b), there are three procedures and two loops. We see that oval nodes represent procedures and circular nodes represent loops. Edges to procedure nodes represent calls while edges to loop nodes shows that the loop is nested in its parent. The program points – namely the starts of procedures and loops – are represented by the start of the code in each oval or circular node. In case of a loop, its program point is outside the loop at its start. In case of a procedure, its program point is inside its body at its start.

Figure 7.4(b) also shows one or more timestamps (e.g 1.1, 1.2) for each node in the DPRG. Since the start of each node is a program point, this timestamps the program points as well. Timestamps are derived using the following rule: the timestamp for each node is the timestamp of its parent appended by a “.” followed by a number representing which child it is in a left to right order. In this way if the `main()` function is assigned a timestamp of 1, the timestamps of all nodes can be computed by a simple variant of the well-known breadth-first-search graph traversal method. The figure shows the results. A node may get more than one timestamp if it has more than one parent. An example of such a scenario is the node for `proc-C()`, which is marked with two timestamps: 1.1.1 and 1.2.1. An ordering on timestamps is their dictionary order. In other words, timestamps are compared according to the following rule: find their longest common prefix ending with a “.”; the larger timestamp is the one with the larger subsequent number. For example, $1.2.1 < 1.3$ since their longest common prefix ending with a “.” is “1.”, and the subsequent number (2) for the first timestamp is less than that of the second timestamp (3). With such an ordering, the timestamps always form a total order among themselves.

Timestamps are useful because they reveal dynamic program execution order: the order in which the program points are visited at runtime is roughly the same as the total order of their timestamps.

This initial set of program points is further refined by considering the actual execution frequency at each program point. We define a threshold `LIMIT` and discard the program points whose execution frequency is greater than `LIMIT`, since the overhead of locking at those points might be too high. The actual value of `LIMIT` is determined through heuristics, as discussed in Section 7.9. Since the timestamps always form a total order among themselves, removing some timestamps from the list does not impact the relative order of remaining timestamps.

Figure 7.4(c) shows the resulting program points and corresponding timestamps after applying the above refinement in this example. Each program point in this modified graph denotes the beginning point of a region. The code block between two consecutive program points is considered a region. As evident, a code block can simultaneously be part of two different regions. The locked content in a code block will be dynamically governed as per the actual execution path.

7.7.2 Dynamic Locking Algorithm

The above method divides a program into a set of regions, where the program locality is consistent in a region. Each region can be analyzed separately since only one region is active at an execution instant. Consequently, the solution for cache locking in each region is obtained by extending the static cache locking algorithm

proposed in Section 7.6 with some modifications, as discussed next.

We introduce the following definitions to ease the description. Similar to static cache locking method, the formulations are for one particular cache set \mathbf{s} .

\mathbf{R} : Set of program regions; \mathbf{r} : An element of set \mathbf{R} ; \mathbf{p} : A program point;

$\text{Exec}_{\mathbf{p}}$: Execution frequency of point \mathbf{p} ; LockInst : Number of cycles for locking a line

$\mathbf{Y}_{\mathbf{r}}$: Set of virtual cache lines accessed in a region \mathbf{r} mapped to a particular cache set \mathbf{s} ¹

$\mathbf{L}_{\mathbf{r}}$: Lines which should be locked in a set \mathbf{s} in a region \mathbf{r}

The dynamic cache locking problem has two objectives (i) For each region \mathbf{r} , determining $\mathbf{L}_{\mathbf{r}}$ (ii) selecting $|\mathbf{L}_{\mathbf{r}}|$ virtual cache lines out of $|\mathbf{Y}_{\mathbf{r}}|$ candidates which should be locked in the set in this region.

The benefit for locking a line in a region \mathbf{r} is same as static cache locking and is given by Equation 7.7. However, the cost for locking a line in a region is influenced by two factors. First, only the program addresses belonging to region \mathbf{r} need to be considered for computing the opportunity cost. Second, the locking instructions are now executed each time a program point is executed. Hence, the cost of locking is no more negligible and the model needs to be reflect the locking cost.

¹Since we are considering solution independently for each cache set, we have simplified the notation by ignoring the set representation \mathbf{s} in the notation for $\mathbf{Y}_{\mathbf{r}}$ and $\mathbf{L}_{\mathbf{r}}$

N : Cache size in number of lines; K : Number of lines which can be locked in each set
 S : Number of sets in the cache.
 R : Set of regions determined by DPRG
 Y_r : The set of memory addresses which get mapped to set s_i in a region r
 $F(x_i)$: Total number of memory accesses to address x_i
 LL : Iterator over number of lines locked in one set in region r
 $HIT_{LL}^r(x_i)$: Total number of hits obtained for x_i when LL lines are locked in set s_i in region r
 $LockList^r(s_i)$: Set of virtual cache lines which should be locked in set s_i in region r
 $NumLockLines^r(s_i)$: Number of virtual cache lines which should be locked in set s_i in region r
 T_{HIT} / T_{MISS} : Hit/Miss latency in processor cycles
 $LockInst$: Number of cycles for locking a lines;
 $Exec_r$: Execution frequency at beginning of region r

```

void Dynamic_Cache_Locking_Algorithm() {
1.  for (each region  $r$  in set  $R$ ) do {
2.    for (each set  $s_i$  in range 0 to  $S - 1$ ) do {
3.      for(each  $LL$  in range 0 to  $K - 1$ ) do {
4.        for (each  $x_i$  in  $Y_r$ ) {
5.           $BenLock(x_i) = (F(x_i) - HIT_{LL}^r(x_i) - 1) * (T_{MISS} - T_{HIT})$ 
6.           $OppCostDynLock(x_i) = \sum_{(x_j | x_j \in Y_r \& x_j \neq x_i)} Cost_{Lock(x_i)}(x_j)$ 
7.           $CostDynLock(x_i) = OppCostDynLock(x_i) + LockInst * Exec_r$ 
8.           $NetBenefit(x_i) = BenLock(x_i) - CostDynLock(x_i)$ 
9.        }
10.       If there exists a  $x_k$  such that  $NetBenefit(x_k)$  is maximum and is positive.{
11.         Add  $x_k$  to  $LockList^r(s_i)$ 
12.          $NumLockLines^r(s_i) = NumLockLines^r(s_i) + 1$ 
13.          $Y_r = Y_r - x_k$ 
14.       }
15.       else {
16.         break; //Locking done for this set
17.       }
18.     }
19.   }
20. }
22.}

```

Figure 7.5: *Dynamic Cache Locking Algorithm.*

The opportunity cost of locking a single element is given by Equation 7.9. However, this opportunity cost is observed only by the elements belonging to region r . Hence, the new opportunity cost is given by the following equation:

$$OppCostDynLock(x_{r_i}) = \sum_{(x_j | x_j \in Y_r \& x_j \neq x_i)} Cost_{Lock(x_i)}(x_j) \quad (7.12)$$

Further, the total cost, considering the actual cost of locking, is defined as follows:

$$CostDynLock(x_{r_i}) = OppCostDynLock(x_{r_i}) + LockInst * Exec_p \quad (7.13)$$

Hence, the net benefit of locking a line can be denoted as

$$NetBenefit(x_{r_i}) = BenLock(x_i) - CostLock(x_{r_i}) \quad (7.14)$$

The above net benefit heuristic is applied in each region to determine the set of lines to be locked in each region. Fig 7.5 presents the pseudo code for dynamic cache locking algorithm.

7.8 Implementation

In this section, we discuss the implementation of binary rewriting scheme for instruction cache locking. Figure 7.6 presents an overview of our experimentation workflow. Our mechanism for cache locking can be implemented inside any existing binary rewriting framework such as Diablo [57] or SecondWrite [10].

First, the binary rewriter framework is employed to obtain an intermediate representation (IR) from the input binary. Next, the techniques presented in Section 7.7.1 are employed to determine the DPRG regions/program points in the input binary. Next, the IR is instrumented with dummy placeholders at these program points and the binary rewriter’s backend is employed to obtain an instrumented binary.

The instrumented binary is used to obtain an instruction trace of the application using a processor simulator (details below). Next, this instruction trace is used to obtain cache statistics using a cache simulator. This cache simulation is iteratively applied with increasing number of lines locked per set to determine the

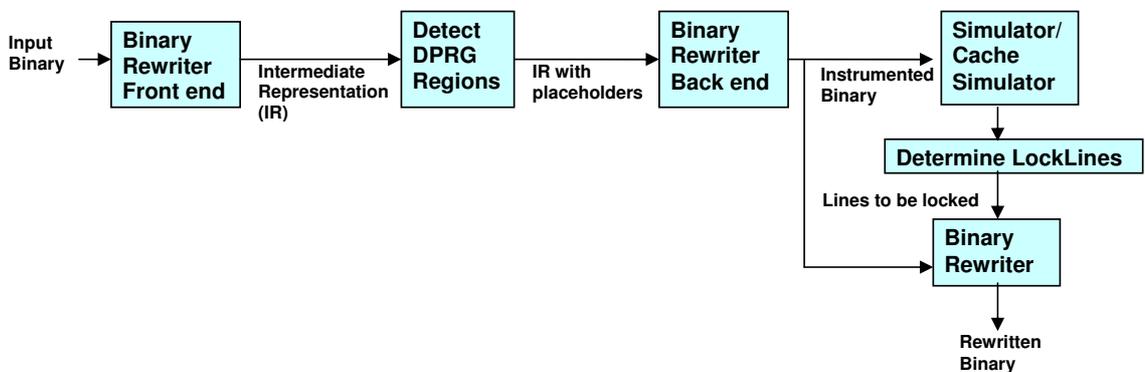


Figure 7.6: *The Experimental WorkFlow*

final list of virtual cache lines to be locked in the cache in each program region.

As mentioned in Section 7.2, special locking instructions are provided in target platforms which upon execution lock the elements at specified addresses in the cache lines. Since our method is based on analyzing the instruction memory addresses of original binary, the code layout of the re-written binary should be exactly the same as instrumented binary. Modifying the program layout might render the above analysis to be incorrect. Hence, after determining the addresses to be locked in the cache, binary rewriter is employed again to insert the actual lock instruction corresponding to the required lines and the remaining placeholders are replaced by NOP instructions.

The workflow in Figure 7.6 corresponds to dynamic locking. In case of static locking, a single placeholder is inserted only at the beginning of the binary, instead of determining the placeholders using DPRG, but otherwise the workflow is similar.

7.9 Results

The experiment setup consists of a Intel XScale processor core with clock frequency 600 Mhz (PXA27x family), on-chip 16 kB 4-way set-associative data cache, on-chip instruction cache and a unified off-chip memory. The ARMulator software, which is part of ARM Development Suite is used to simulate the processor core. The above architectural parameters can be easily configured in ARMulator. Dinero IV [60], a well-known cache-hierarchy simulator is used to simulate the cache. We modified Dinero to provide the cache statistics at the granularity of virtual cache lines and

Application	Lines Of Code	Num of Instr	Num of DynInstr
Sha	207	2501	355452842
Crc	128	1027	75738737
BitCnts	543	3340	149409187
Susan	1456	4040	60516192
Blowfish	3260	2909	868261350
Jpeg	19804	9718	104615385
Dijkstra	268	18612	536074136
Lame	15959	19810	569002359
Gsm	4779	14040	64340338
StringSearch	3072	1839	8051466
QuickSort	79	2298	830913008
Lout	30689	59828	538663
FFT	278	5868	671496345
BasicMath	7367	6375	102147075
Patricia	296	7756	114446172
Rinjdal	1017	4960	578559602

Table 7.1: *Application Table*

augmented it with the ability to simulate cache locking.

We configured the ARMulator to simulate a perfect zero-wait memory system. It generates the execution time in terms of processor cycles. The instruction and data miss statistics provided by Dinero are used to calculate the effect of cache misses on execution time and is added to the execution time calculated by ARMulator to obtain the total execution time of the application. A sample memory map file available in ARMulator with average off-chip memory access time of 150ns is chosen to calculate off-chip memory access latency. As per the XScale’s architecture

manual, each locking instruction is assumed to take four cycles and is considered accordingly while calculating the execution time of resulting binary.

A subset of MiBench benchmarks were randomly selected to substantiate the performance improvement obtained by our method of cache locking. At this point we have simply included all the benchmarks that compiled and ran in our infrastructure in the time available – the benchmarks have not been selected to be favorable to us in any way. Table 7.1 lists the benchmarks which are used for carrying out the experiments. All the benchmarks are statically compiled with the GNU-ARM toolchain.

The results for static cache locking are presented in Section 7.9.1 while Section 7.9.2 presents the improvement obtained by dynamic cache locking over the static mechanism. Section 7.9.2 also compares our static and dynamic mechanisms with the static mechanism suggested by Liang and Mitra [98]. We refer to the method suggested in [98] as `OPT-static`. Based on the execution frequency of program regions, the value of `LIMIT` (Section 7.7.1) was kept to 50 in our experiments.

7.9.1 Static Cache Locking

Various kinds of experiments are performed with different cache configurations for analyzing the improvement in the instruction-cache miss rate and run-time of the applications. The cache configuration is varied across two dimensions: size and associativity. The block size is kept fixed at 4 words.

The percentage improvement in the I-cache miss rate with static cache locking

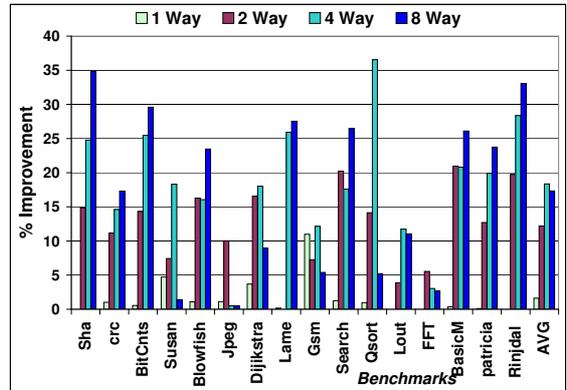
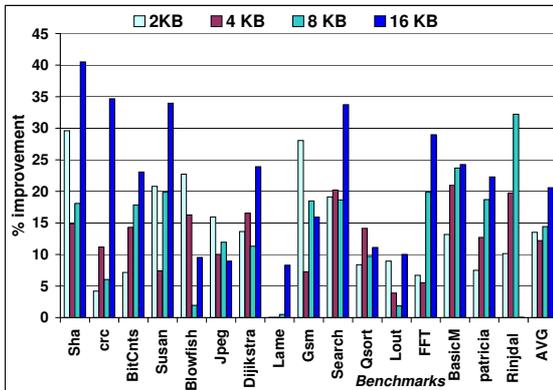


Figure 7.7: Percentage improvement in instruction-cache miss rate over cache with no locking for varying sizes of a 2-way set-associative cache

Figure 7.8: Percentage improvement in instruction-cache miss rate over cache with no locking for different associativities of the cache. The cache size is kept fixed at 4 Kb.

compared to the cache configuration without locking is displayed in Figure 7.7 for different cache sizes. As evident from this figure, the proposed I-cache locking mechanism results in a consistent improvement in the instruction cache miss rate over all the benchmarks and cache sizes. We obtain an average improvement of 15% in the I-cache miss rate for small cache sizes and around 25% for large cache sizes. Interestingly, the improvement in the I-cache miss rate increases with an increase in the cache size for most of the applications. This is expected since a small cache size results in a high opportunity cost in our cost-benefit model as locking a line prevents many other lines from accessing that cache location, resulting into fewer lines being locked in the cache.

Figure 7.8 displays the variation of I-cache miss rate improvement with variation in associativity of the cache. We see that the improvement in the I-cache miss rate ranges from 15-18% for set-associative caches. Virtually all commercial

cached embedded processors support only set-associative caches², which establishes our proposed approach as a robust mechanism for improving memory system performance. The average improvement in case of direct mapped cache is, not surprisingly, limited – having only one way in a set amounts to extremely high opportunity cost resulting in very little locking. Our goal is not to get improvements in direct-mapped cache – we never expected to, and such caches are very rare in embedded systems – but the results are presented for completeness, and show that the method never degrades performance, even managing a small improvement for direct mapped caches³.

Next, the impact of instruction cache locking on run-time performance of various applications is analyzed. Figure 7.9 shows the savings in runtime obtained by using the instruction cache-locking. Comparing Figure 7.7 and Figure 7.9 brings out several interesting observations.

First, even though the cache locking scheme results in considerable improvement of instruction cache miss rate consistently over all the applications, not all applications experience an improvement in run-time performance. The improvement in I-cache miss rates translates to run-time performance improvement only for those applications where the overall I-cache miss rate is high. This is not surprising since a technique like ours to reduce the I-cache miss rate will not help if it is not a problem to begin with.

Revisiting Figure 7.9, we see that the benchmarks on the right-hand side are

²For example, among the ARM processors, only one of the 15 processors listed on ARM’s website offers a direct-mapped cache.

³For simulation purposes, the architectural constraint of not locking way 0 is relaxed for direct mapped cache.

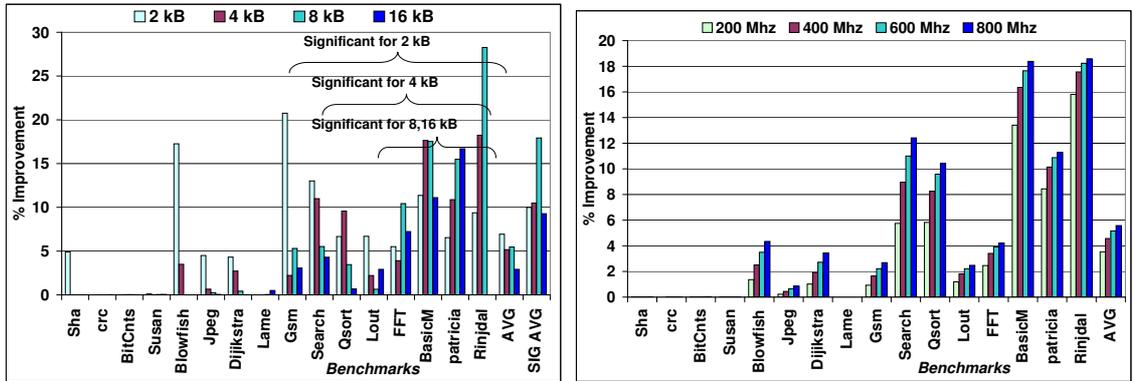


Figure 7.9: *Improvement in execution time of the applications over cache with improvement for processors with different no locking for varying size of a 2-way set associative cache* Figure 7.10: *Variation of execution time of the applications over cache with improvement for processors with different no locking for varying size of a 2-way set associative cache*

marked “significant” for different cache sizes. These are the benchmarks that have a significant I-cache miss rate (which we define as $> 1.5\%$) for that cache size. For the benchmarks with significant I-cache miss rate, the run-time improvement from our cache locking method averages 11.5% for a cache size of 8kB. The averages are shown in Figure 7.9 in the last two columns as AVERAGE and SIG-AVERAGE, for all the benchmarks, and those with significant miss rates, respectively. For the benchmarks with very low I-cache miss rates, the benefits from cache locking are, not surprisingly, low – only 1.7 % for a 2kB cache.

The 11.5% run-time improvement with cache locking for benchmarks with a significant I-cache miss rate is encouraging and shows the benefit of our method. For some benchmarks the benefit is even higher – e.g, the Rinjidal benchmark has a run-time gain of 23%. Overall we see that about 20-60% of the benchmarks show significant improvement, depending on the cache size and associativity used. The fact that not all benchmarks benefit from cache locking is not an indictment against our method – indeed there is a long history of research into techniques that benefit

only a class of applications ⁴. As classes of applications go, benefiting 20-60% of benchmarks significantly is quite good.

Further, we observe that although increasing cache size results in better performance in terms of instruction cache miss rate reduction, the average percentage improvement in execution time decreases with an increase in cache size. An increase in the cache size results in a lower initial miss rate and thus yields smaller run-time benefits from locking.

Next, in order to analyze the applicability of our approach for different processor generations, we analyze the improvement in execution time for various processor frequencies. We vary the processor clock speed while keeping the DRAM latency constant in nanoseconds – this is equivalent to varying the DRAM latency in cycles. We obtain a consistent improvement in execution time with an increase in processor frequency, as displayed in Figure 7.10. Thus our method can be applied effectively for different generations of processors.

7.9.2 Dynamic Cache Locking

In this section, we present the results for our dynamic cache locking algorithm and compare the results with our static algorithm (`Static` in `CacheLocking/figures`) as well as the `OPT-static` algorithm suggested by Liang and Mitra [98].

Figure 7.11 presents the percentage improvement in the I-cache miss rate with dynamic cache locking, as compared to both static algorithms, for different cache

⁴e.g. faster garbage collectors only benefit benchmarks with heap data, and among those, only those with significant garbage – however garbage collection is still worthwhile.

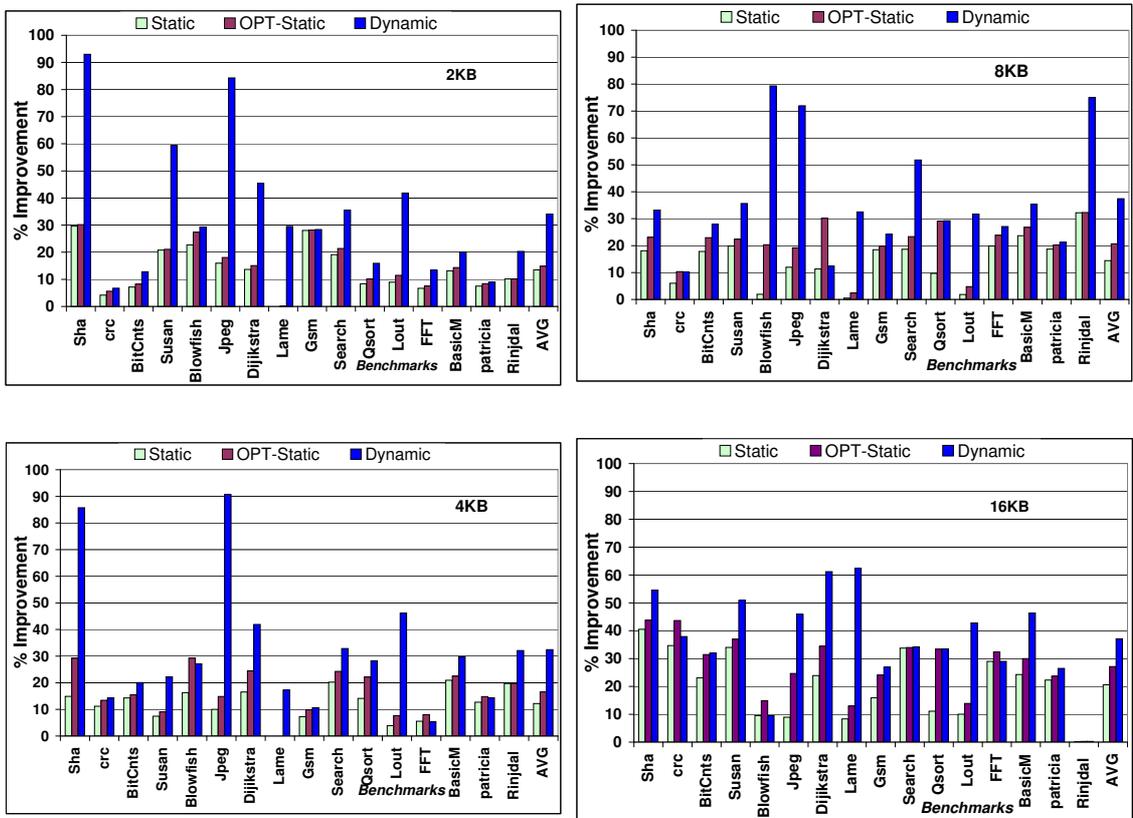


Figure 7.11: Percentage improvement in instruction-cache miss rate, compared with static cache locking, for a 2-way set-associative cache of size 2 kb, 4 kb, 8 kb and 16 kb.

sizes. As evident from this figure, the dynamic I-cache locking mechanism results in a consistent improvement in the instruction cache miss rate over all the benchmarks and cache sizes. We obtain an average improvement in the range of 35% to 40% in the I-cache miss rate for all cache sizes.

Figure 7.11 shows that OPT-static obtains a slightly better I-cache miss rate than our static algorithm, thereby revalidating the results presented in [98]. However, our dynamic mechanism consistently results in a better I-cache miss rate than both static methods. We also notice a few scenarios (blowfish - 16 kb, FFT - 8 kb) where our dynamic version performs worse than OPT-static algorithm. We believe

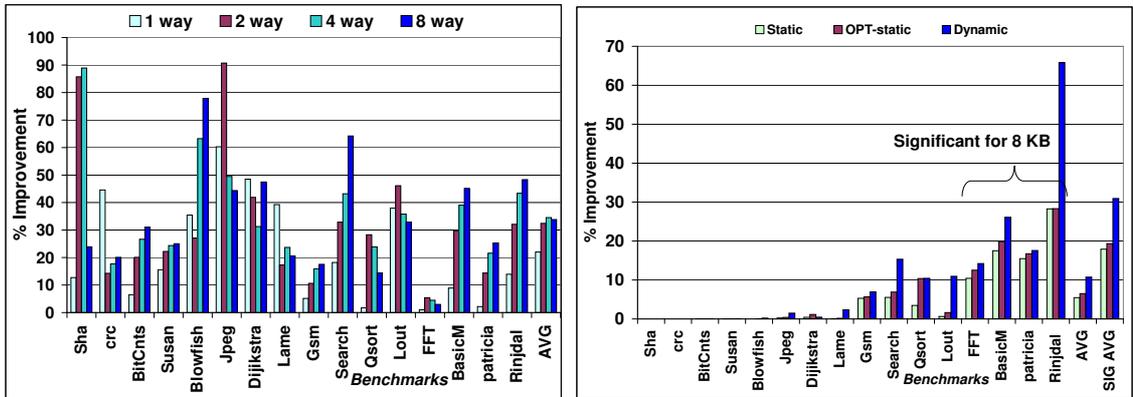


Figure 7.12: Percentage improvement in instruction-cache miss rate with dynamic cache locking over cache with no locking for different associativities of the cache. The cache size is kept fixed at 8 kb.

Figure 7.13: Improvement in execution time of the applications with dynamic cache locking, as compared with static and optimal static cache locking, for varying size of a 8 kb 2-way set associative cache.

this can be overcome by actually applying OPT-static, instead of applying our static version, to determine the lines to be locked within each program region.

The results in Section 7.9.1 demonstrate that the improvement in the I-cache miss rate due to static cache locking increases with an increase in the cache size for most of the applications. Figure 7.11 demonstrates that dynamic cache locking mechanism does not display this behavior. It results in a consistent improvement of around 40% across all cache sizes. This is not surprising since the dynamic mechanism overcomes the inherent opportunity cost involved in the static locking mechanism by dynamically adapting the cache content with program demand. An interesting corollary of this result is that the dynamic mechanism is much more effective for smaller cache sizes. For example, for cache size of 4 kB, the dynamic method improves the cache miss rate by 35% as compared to 15% by static method whereas in case of a 16 kB cache, the relative improvement is 37% over 27% improvement

obtained by static mechanism.

Figure 7.12 displays the variation of I-cache miss rate improvement with variation in associativity of the cache. We see that the improvement in the I-cache miss rate due to dynamic cache locking ranges from 32-35% for different associativity of set associative caches, as compared to 15-18% improvement obtained by static mechanisms. An interesting feature is that dynamic cache locking is also able to obtain 20% performance improvement for direct mapped caches. Recall from Section 7.9.1, static cache locking was mainly effective for set associative caches. This is due to the reduced opportunity cost in dynamic locking models.

Next, the impact of instruction cache locking on run-time performance of various applications is analyzed. Figure 7.13 shows the reduction in runtime by using dynamic instruction cache-locking for a particular cache configuration, as compared to static algorithms. Similar to Fig 7.9, we average the improvement in execution time for all the benchmarks as well as the benchmarks with significant initial miss rate. Figure 7.13 demonstrates that the improvement in miss rate obtained by dynamic algorithms translates effectively to an improvement in execution time. For benchmarks with a significant I-Cache miss rate, the dynamic mechanism improves execution time by 20% on average as compared to 11.5% and 12.5% obtained by static and OPT-static mechanisms respectively.

Chapter 8: **Conclusions and Future Work**

It is conventional wisdom that static analysis of executables is a very difficult problem, resulting in a plethora of dynamic binary frameworks. However, a static binary framework based on a compiler IR enables applications not possible in any existing tool and our results establish the feasibility of this approach for several pragmatic scenarios. We do not claim that we have fully solved all the issues; statically handling every program in the world may still be an elusive goal. However, the resulting experience of expanding the static envelope as much as possible is a hugely valuable contribution to the community.

In this work, we have presented several component techniques essential for translating executables to a high-level intermediate representation of an existing compiler. Our techniques overcome challenges unique to executables: an explicitly addressed stack, the lack of function prototypes and the lack of symbols. The compiler IR allows the application of source-level complex transformations and advanced symbolic execution strategies on executables and enables functional source-code recovery.

Next, we have proposed techniques to obtain a functional and precise representation from executables and presented methods to adapt symbolic analysis to work

effectively on executables. The improved memory model considerably enhances the precision of our symbolic analysis framework and our symbolic analysis framework improves the efficacy of various analyses.

We extend our underlying representation and analysis framework to define DemandFlow, a novel information-flow framework for executables, which possesses the desired properties of practicality, precision, scalability and extensibility. DemandFlow uncovers seven new zero-day vulnerabilities in popular programs at a false positive rate comparable to source-level tools.

We demonstrate another application of our framework by formulating an instruction cache locking mechanism in a binary rewriter. We present the first method in literature for improving the average-case run-time of embedded systems, extending the applicability of cache locking beyond real-time systems. Our results indicate that on average, the proposed cache locking scheme achieves a 32% improvement in run-time performance of instruction cache-constrained applications.

8.1 Future Directions

Figure 8.1 presents several possible future extensions of our work. The future extensions are categorized analogous to the contributions of this dissertation.

First, several enhancements can be made in expanding the scope of our representation recovery mechanism. Our current techniques do not handle self modifying code or dynamically generated code. We have not tested our techniques against executables with hand-coded assembly or with obfuscated control flow. Further,

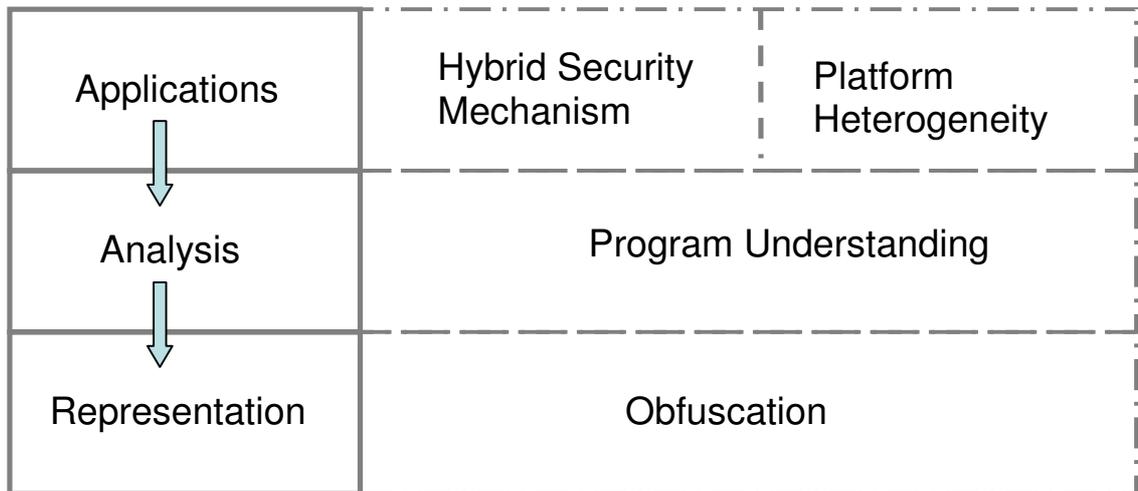


Figure 8.1: *Future Directions*

SecondWrite’s underlying disassembler [135] distinguishes code and data using restrictions of a compiled code.

It is very difficult for any analysis tool to statically reason about the code generated at runtime. Hence, a natural extension would be to define a hybrid static-dynamic framework to recover intermediate representation in presence of obfuscated and dynamically generated code. Several dynamic mechanisms [5] have been proposed in this regard. Such dynamic mechanisms can be integrated with our static frontend to recover a detailed representation.

Next, given a possible untrusted executable program, a rudimentary task is to determine the behavior and purpose of the program. There has been extensive research in architecture recovery of software systems by parsing its source code [41, 105, 51]. Several software engineering methods have been proposed to identify the modular components of a system and to discover the relations between such components. Such architecture recovery methods can be applied on our analysis

framework for recovering an abstract description of a unknown executable sample. This will enhance the ability to uncover functionality and possible unsafe behavior of an unknown executable.

Our current security vulnerability mechanism, DemandFlow, is a static mechanism and does report a few false positives. In scenarios where the reporting of false positives may be considered unacceptable, the analysis can be aided with a hybrid static-dynamic framework.

Finally, we believe that the ability to obtain a functionally equivalent rewritten executable can be employed to solve an important challenge posed by increasing heterogeneity of computing platforms. The increasing heterogeneity of modern processors has posed a serious problem to the standard software development tool-chain. Most of the users do not develop software programs and just install third party software on their systems. The third party softwares are distributed in the form of compiled binaries as revealing the source code would reveal the IP. The software developers are not aware of the details of the platform where the software is going to run. Different processors employ different level of features like number of cores, hardware pipeline, memory systems, functional units and these features cannot be fully exploited by an existing compiler tool-chain due to the unavailability of such information while compiling the software. This results in a widening gap between peak processor performance and sustained processor performance, and software is able to exploit only a fraction of the available performance on the processor. This processor performance gap can be bridged by adapting the software to the user platform using SecondWrite.

Bibliography

- [1] *Directory traversal vulnerability in Rack*. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0262>.
- [2] Idapro disassembler. <http://www.hex-rays.com/idapro/>.
- [3] Keepassx - cross platform password manager. <http://www.keepassx.org/>.
- [4] Oink tool. <http://daniel-wilkerson.appspot.com/oink/index.html/>.
- [5] Ollydbg debugger. <http://www.ollydbg.de/>.
- [6] B. Alpern and et. al. The jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.
- [7] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '88*, pages 1–11, New York, NY, USA, 1988. ACM.
- [8] W. Amme, P. Braun, F. Thomasset, and E. Zehendner. Data dependence analysis of assembly code. *Int. J. Parallel Program.*, 28(5):431–467, Oct. 2000.
- [9] K. Anand and R. Barua. Instruction cache locking inside a binary rewriter. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '09*, pages 185–194, New York, NY, USA, 2009. ACM.
- [10] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13*, pages 295–308, New York, NY, USA, 2013. ACM.
- [11] Announcement for Binary Executable Transforms. <http://www07.grants.gov/>.

- [12] Apache HTTP server benchmarking tool. <http://httpd.apache.org/docs/2.2/programs/ab.html>.
- [13] Application Security testing - Veracode. <http://www.zynamics.com//>.
- [14] *ARM1156T2-S Technical Reference Manual*. Arm, Revised July 2007. <http://www.arm.com/products/CPUs/families/ARM11Family.html>.
- [15] *ARM Cortex A-8 Technical reference manual*. Arm, Revised March 2004. <http://www.arm.com/products/CPUs/families/ARMCortexFamily.html>.
- [16] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RNTS)*, Poitiers, France, May 2006.
- [17] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *S&P, 2002*, SP '02, 2002.
- [18] O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems, Special Issue on Memory Systems*, 1(1), September 2002.
- [19] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86:a platform for analyzing x86 executables. In *Proceedings of the 14th international conference on Compiler Construction*, CC'05, pages 250–254, Berlin, Heidelberg, 2005. Springer-Verlag.
- [20] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, pages 5–23. Springer-Verlag, 2004.
- [21] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *In CC*, pages 5–23. Springer-Verlag, 2004.
- [22] G. Balakrishnan and T. Reps. DIVINE: discovering variables in executables. In *Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation*, pages 1–28, 2007.
- [23] D. Balzarotti, M. Cova, V. Felmetger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *S & P, 2008*, pages 387–401.
- [24] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratch-pad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, May 6-8 2002. ACM.
- [25] U. Banerjee. *Speedup of ordinary programs*. PhD thesis, Champaign, IL, USA, 1979. AAI8008967.

- [26] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.
- [27] L. Belady. A study of replacement algorithms for virtual storage. In *IBM Systems Journal*, pages 5:78–101, 1966.
- [28] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001.
- [29] K. Beyls and E. H. D’Hollander. Generating cache hints for improved program efficiency. *J. Syst. Archit.*, 51(4):223–250, 2005.
- [30] *ADSP-BF533 Processor Hardware Reference*. Analog Devices, April 2009. http://www.analog.com/static/imported-files/processor_manuals/bf533_hwr_Rev3.4.pdf.
- [31] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, 1994.
- [32] R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’98, pages 237–251, New York, NY, USA, 1998. ACM.
- [33] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.
- [34] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. 2008.
- [35] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *CAV*, pages 463–469, 2011.
- [36] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV’11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.
- [37] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.
- [38] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.
- [39] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, 2006.

- [40] A. M. Campoy, A. P. Jimenez, A. P. Ivars, and J. V. B. Mataix. Using genetic algorithms in content selection for locking-caches, 2001.
- [41] G. CanforaHarman and M. Di Penta. New frontiers of reverse engineering. In *2007 Future of Software Engineering, FOSE '07*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.
- [42] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394, 2012.
- [43] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. *CCS '08*, pages 39–50.
- [44] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 416–419, New York, NY, USA, 2000. ACM.
- [45] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 5th European conference on Computer systems*, pages 167–180, 2010.
- [46] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 211–216, 2011.
- [47] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ASPLOS XVI*, pages 265–278, New York, NY, USA, 2011. ACM.
- [48] C. Cifuentes and M. V. Emmerick. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, 2000.
- [49] C. Cifuentes and M. V. Emmerik. Uqbt: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, 2000.
- [50] J. Clause, W. Li, and R. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, pages 196–206, 2007.
- [51] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *IN PROCEEDINGS OF THE 22ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, pages 439–448. ACM Press, 2000.
- [52] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

- [53] M. Cova, V. Felmetzger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC '06*, pages 269–278, Washington, DC, USA, 2006. IEEE Computer Society.
- [54] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.
- [55] C. Cowan and et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium*, pages 63–78, 1998.
- [56] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, 35(2):482–493, June 2007.
- [57] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27(5):882–945, Sept. 2005.
- [58] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, New York, NY, USA, 1998. ACM.
- [59] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *POPL, 1995*, pages 37–48, 1995.
- [60] J. Edler and M. Hill. Dineroiv cache simulator. Revised 2004. <http://www.cs.wisc.edu/markhill/DineroIV/>.
- [61] M. Egele and et al. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.
- [62] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, PLDI '13*, pages 51–60, New York, NY, USA, 2013. ACM.
- [63] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. University of maryland technical report - recovering function boundaries from executables. 2013. <http://www.ece.umd.edu/barua/function-boundaries.pdf>.
- [64] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

- [65] H. ETO and K. Yoda. Propolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer Security 14 (Oct 26)*, pages 4034–4041, 2001.
- [66] A. Eustace and A. Srivastava. ATOM: a flexible interface for building high performance program analysis tools. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference*, pages 25–25, 1995.
- [67] T. Fahringer and B. Scholz. Symbolic evaluation for parallelizing compilers. In *International Conference on Supercomputing*, pages 261–268, 1997.
- [68] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 143–148, New York, NY, USA, 2007. ACM.
- [69] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, PLDI '91*, pages 15–29, New York, NY, USA, 1991. ACM.
- [70] A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting fixed programs in embedded systems: A loop cache example. *IEEE Comput. Archit. Lett.*, 1(1):2, 2002.
- [71] S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *POPL*, pages 74–84, 2003.
- [72] S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *POPL*, pages 342–352, 2004.
- [73] S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '04*, pages 342–352, New York, NY, USA, 2004. ACM.
- [74] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and accurate low-level pointer analysis. In *Proceedings of the international symposium on Code generation and optimization, CGO '05*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.
- [75] S. Z. Guyer and C. Lin. Client-driven pointer analysis. *SAS'03*, pages 214–236.
- [76] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 18(4):477–518, July 1996.
- [77] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, July 2005.

- [78] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, 3(3):243–250, May 1977.
- [79] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI, 2001*.
- [80] Hex-Rays Decompiler. <http://www.hex-rays.com/>.
- [81] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. Scalable High Performance Computing Conference, May 1994.
- [82] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *Comput. J.*, 23(3):223–229, 1980.
- [83] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.
- [84] IDAPro disassembler. <http://www.hex-rays.com/idapro/>.
- [85] K. Jee and et al. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *NDSS, 2012*.
- [86] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *S & P, 2006*, pages 258–263.
- [87] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical Data Flow Tracking for Commodity Systems. In *VEE, 2012*, pages 121–132.
- [88] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.
- [89] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag.
- [90] A. King, A. Mycroft, T. Reps, and A. Simon. Analysis of Executables: Benefits and Challenges. *Dagstuhl Reports*, pages 100–116, 2012.
- [91] A. King, A. Mycroft, T. W. Reps, and A. Simon. Analysis of executables: Benefits and challenges (dagstuhl seminar 12051). *Dagstuhl Reports*, 2(1):100–116, 2012.
- [92] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. *ICISS '08*, pages 56–70.

- [93] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '10*, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society.
- [94] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. *ACSAC 2006*, pages 463–472.
- [95] J. R. Larus and E. Schnarr. Eel: machine-independent executable editing. *SIGPLAN Not.*, 30(6):291–300, June 1995.
- [96] 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*, pages 75–87, 2004.
- [97] J. Li, C. Wu, and W.-C. Hsu. Dynamic register promotion of stack variables. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 21–31, 2011.
- [98] Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference, DAC '10*, pages 344–349, New York, NY, USA, 2010. ACM.
- [99] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security, CCS '03*, pages 290–299, New York, NY, USA, 2003. ACM.
- [100] T. Liu, M. Li, and C. J. Xue. Instruction cache locking for embedded systems using probability profile. *J. Signal Process. Syst.*, 69(2):173–188, Nov. 2012.
- [101] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14, SSYM'05*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.
- [102] C.-K. Luk and et al. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM conference on Programming language design and implementation*, pages 190–200, 2005.
- [103] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A post-link optimizer for the Intel Itanium architecture. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 15–26, 2004.
- [104] D. Marino and et. al. A case for an sc-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11*, pages 199–210, New York, NY, USA, 2011. ACM.

- [105] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 32:193–208, 2006.
- [106] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL, 1999*, pages 228–241.
- [107] S. Nanda and et al. Bird: Binary interpretation using runtime disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, Washington, DC, USA, 2006.
- [108] NAS Parallel Benchmarks. <http://www.nas.nasa.gov/publications/npb.html/>.
- [109] New unique samples added to AV-Test Malware Repository. <http://www.eset.com/us/threat-center/>.
- [110] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS05*, 2005.
- [111] P. OSullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. Keromytis. Retrofitting security in cots software with binary rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 154–172. Springer Berlin Heidelberg, 2011.
- [112] P. R. Panda, N. D. Dutt, and A. Nicolau. On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3), July 2000.
- [113] J. R. C. Patterson. Accurate static branch prediction by value range propagation. *SIGPLAN Not.*, 30(6):67–78, June 1995.
- [114] Phoenix Compiler Infrastructure. <http://www.research.microsoft.com/phoenix/>.
- [115] The Polyhedral Benchmark Suite. <http://www.polyhedron.com/MFL6VW74649/>.
- [116] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proc. of the 2nd International Workshop on worst-case execution time analysis, in conjunction with the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.
- [117] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the 23rd IEEE International Real-Time Systems Symposium*, Austin, TX, USA, December 2002.
- [118] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

- [119] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39*, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.
- [120] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *In Symposium on Principles of Programming Languages*, pages 119–132, 1999.
- [121] X. Rival. Abstract interpretation-based certification of assembly code. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2003*, pages 41–55, London, UK, UK, 2003. Springer-Verlag.
- [122] T. Romer and et al. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *In Proceedings of the USENIX Windows NT Workshop*, pages 1–1, 1997.
- [123] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '88*, pages 12–27, New York, NY, USA, 1988. ACM.
- [124] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *SIGPLAN Not.*, 35(5):182–195, May 2000.
- [125] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.
- [126] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang. Cooperative caching with keep-me and evict-me. In *INTERACT '05: Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 46–57, Washington, DC, USA, 2005. IEEE Computer Society.
- [127] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis, ISSTA '09*, pages 225–236, New York, NY, USA, 2009. ACM.
- [128] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO, 2008*, pages 74–83, 2008.
- [129] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P, 2010*, pages 317–331, 2010.

- [130] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *In Proc. Workshop on Binary Translation*, 2001.
- [131] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-linux. <http://developer.kde.org/~sewardj/>.
- [132] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium, 2001*, pages 16–16.
- [133] Simplex method in IDA Pro. <http://www.hexblog.com/?p=42>.
- [134] J. Sjodin, B. Froderberg, and T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, December 1998.
- [135] M. Smithson and R. Barua. Binary Rewriting without Relocation Information. *USPTO patent pending no. 12/785,923*, May 2010.
- [136] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security, ICISS '08*, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.
- [137] D. Song and et al. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, pages 1–25, 2008.
- [138] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, page 409. IEEE Computer Society, 2002.
- [139] Symantec. *Symantec Internet Security Technical Report*, volume 17 edition, 2011. <http://www.symantec.com/threatreport/>.
- [140] O. Temam. Investigating optimal local memory performance. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems, ASPLOS VIII*, pages 218–227, New York, NY, USA, 1998. ACM.
- [141] A. Thakur and T. Reps. A method for symbolic computation of abstract operations. *CAV'12*, pages 174–192, 2012.
- [142] R. A. Towle. *Control and data dependence for program transformations*. PhD thesis, Champaign, IL, USA, 1976. AAI7624191.

- [143] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *In PLDI, 2009*, pages 87–97.
- [144] P. Tu and D. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th international conference on Supercomputing, ICS '95*, pages 414–423, New York, NY, USA, 1995. ACM.
- [145] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 276–286. ACM Press, 2003.
- [146] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.
- [147] 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 *MICRO 2004*, pages 243–254.
- [148] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, 2007.
- [149] L. van Put, D. Chanet, B. De Bus, B. De Sutler, and K. De Bosschere. DIA-BLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, 2005.
- [150] T. Vandrunen and A. L. Hosking. Value-based partial redundancy elimination. In *In CC*, pages 167–184, 2004.
- [151] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, New York, NY, USA, 2003. ACM.
- [152] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, automation and test in Europe*, page 21264. IEEE Computer Society, 2004.
- [153] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratch-pad memory for energy minimization. In *International conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS)*. ACM, 2004.

- [154] D. Wagner and et. al. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.
- [155] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.
- [156] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. STILL: Exploit code detection via static taint and initialization analyses. In *Computer Security Applications Conference, Annual*, pages 289–298, 2008.
- [157] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.
- [158] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, pages 179–192, 2006.
- [159] *3rd Generation Intel Xscale Microarchitecture Developer’s manual*. Intel, May 2007. <http://www.intel.com/design/intelxscale/>.
- [160] W. Xu, E. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.
- [161] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu. Improving power efficiency with compiler-assisted cache replacement. *J. Embedded Comput.*, 1(4):487–499, 2005.
- [162] E. Yardimci and M. Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd conference on Computing frontiers*, CF ’06, pages 127–138, New York, NY, USA, 2006. ACM.
- [163] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS ’07, pages 116–127, New York, NY, USA, 2007. ACM.
- [164] J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 501–508, 2007.
- [165] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM international symposium on Microarchitecture*, pages 85–96, 2002.