

AUTOMATIC DETECTION OF SAFETY AND SECURITY  
VULNERABILITIES IN OPEN SOURCE SOFTWARE

SYRINE TLILI

A THESIS  
IN  
THE DEPARTMENT  
OF  
ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

CONCORDIA UNIVERSITY  
MONTRÉAL, QUÉBEC, CANADA

AUGUST 2009

© SYRINE TLILI, 2009



Library and Archives  
Canada

Published Heritage  
Branch

395 Wellington Street  
Ottawa ON K1A 0N4  
Canada

Bibliothèque et  
Archives Canada

Direction du  
Patrimoine de l'édition

395, rue Wellington  
Ottawa ON K1A 0N4  
Canada

*Your file* *Votre référence*  
ISBN: 978-0-494-63416-5  
*Our file* *Notre référence*  
ISBN: 978-0-494-63416-5

**NOTICE:**

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

**AVIS:**

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

---

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

  
**Canada**

# ABSTRACT

## Automatic Detection of Safety and Security Vulnerabilities in Open Source Software

Syrine Tlili, Ph.D.

Concordia University, 2009

Growing software quality requirements have raised the stakes on software safety and security. Building secure software focuses on techniques and methodologies of design and implementation in order to avoid exploitable vulnerabilities. Unfortunately, coding errors have become common with the inexorable growth tendency of software size and complexity. According to the US National Institute of Standards and Technology (NIST), these coding errors lead to vulnerabilities that cost the US economy \$60 billion each year. Therefore, tracking security and safety errors is considered as a fundamental cornerstone to deliver software that are free from severe vulnerabilities. The main objective of this thesis is the elaboration of efficient, rigorous, and practical techniques for the safety and security evaluation of source code. To tackle safety errors related to the misuse of type and memory operations, we present a novel type and

effect discipline that extends the standard C type system with safety annotations and static safety checks. We define an inter-procedural, flow-sensitive, and alias-sensitive inference algorithm that automatically propagates type annotations and applies safety checks to programs without programmers' interaction. Moreover, we present a dynamic semantics of our C core language that is compliant with the ANSI C standard. We prove the consistency of the static semantics with respect to the dynamic semantics. We show the soundness of our static analysis in detecting our targeted set of safety errors. To tackle system-specific security properties, we present a security verification framework that combines static analysis and model-checking. We base our approach on the GCC compiler and its GIMPLE representation of source code to extract model-checkable abstractions of programs. For the verification process, we use an off-the-shelf pushdown system model-checker, and turn it into a fully-fledged security verification framework. We also allow programmers to define a wide range of security properties using an automata-based specification approach. To demonstrate the efficiency and the scalability of our approach, we conduct extensive experiments and case studies on large scale open-source software to verify their compliance with a representative set of the CERT standard secure coding rules.

# Acknowledgments

I would like to express my gratitude to Almighty ALLAH, the most Beneficent and the most Merciful, for granting me the ability and opportunity to complete this thesis.

I would like to thank to my supervisor, Dr. Mourad Debbabi, for his advices, ideas and efforts to ensure a continuous supervision of this thesis. His insights and encouragements have had a major impact on this work, which would not be possible without his guidance and support. He deserves all my acknowledgements.

I would like to thank Dr. Ettore Merlo, Dr. Joey Paquet, Dr. Roch Glitho, and Dr. Zhu Bo who honored me by being members of the examiner committee and reviewing this thesis. Their time and effort are greatly appreciated.

I would also like to thank my colleagues Rachid Hadjidj and XiaoChun Yang for their collaboration in designing and implementing the security verification toolkit.

I thank my friends Rabeb Mizouni, Lamia Ketari, Anis Ouali, Hadi Otrok, Azzam Mourad, Yosr Jarraya, Nadia Belblidia, and Dima Alhadidi who shared with me the precious years of my thesis. Also, a very special thank is due to all my TFOSS team colleagues for all their collaboration in this research. Moreover, further thanks are extended to all the members of the Computer Security Laboratory.

Last but by no means least, I am very grateful to my father, mother, brothers, and family members for their love, permanent encouragement, belief in me, and endless support. This work, and my life, would never have been the same without them.

# DEDICATION

To my parents who are my first and best teachers,

To my brothers, who are best friends to me,

To all my family, who permanently believe in me.

# Contents

<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xvi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivations and Problem Statement . . . . .	3
1.1.1 Need for Automated Verification Tools . . . . .	3
1.1.2 Heavy Annotation Burden on Programmers . . . . .	5
1.1.3 Modular Security Property Specification . . . . .	6
1.1.4 Efficient Synergy for Hybrid Approaches . . . . .	7
1.1.5 Flexible Integration of Verification Techniques . . . . .	8
1.2 Objectives . . . . .	9
1.3 Contributions . . . . .	10
1.3.1 Type and Effect Discipline for C Safety . . . . .	10
1.3.2 Static Detection of Runtime Errors . . . . .	12
1.3.3 Verification of Secure Coding Rules . . . . .	13
1.4 Thesis Outline . . . . .	14



<b>2</b>	<b>Survey of Security and Safety Analyses</b>	<b>16</b>
2.1	Introduction . . . . .	16
2.2	What Causes Safety/Security Vulnerabilities ? . . . . .	17
2.2.1	Memory Management Vulnerabilities . . . . .	17
2.2.2	String Manipulation Vulnerabilities . . . . .	23
2.2.3	Race Conditions . . . . .	26
2.2.4	File Management Vulnerabilities . . . . .	27
2.2.5	Privilege Management Vulnerabilities . . . . .	29
2.3	Vulnerability Detection Techniques . . . . .	31
2.3.1	Static Analysis . . . . .	31
2.3.2	Model-Checking . . . . .	36
2.3.3	Dynamic Analysis . . . . .	37
2.4	Comparative Study . . . . .	38
2.4.1	Static Analysis vs. Dynamic Analysis . . . . .	38
2.4.2	Static Analysis vs. Model-Checking . . . . .	41
2.5	Conclusion . . . . .	43
<b>3</b>	<b>Survey of Security and Safety Tools</b>	<b>45</b>
3.1	Annotation-Based Techniques . . . . .	46
3.1.1	Lint Family . . . . .	46
3.1.2	CQual . . . . .	49
3.2	Automata-Based Techniques . . . . .	52

3.2.1	Meta-Compilation . . . . .	52
3.2.2	MOPS . . . . .	55
3.3	Hybrid Techniques . . . . .	57
3.3.1	CCured . . . . .	58
3.3.2	SafeC . . . . .	60
3.4	Comparative Study . . . . .	62
3.4.1	Flow-Sensitive vs. Flow-Insensitive . . . . .	62
3.4.2	Interprocedural vs. Intraprocedural Analysis . . . . .	64
3.4.3	Internal Checking vs. External Checking . . . . .	65
3.5	Conclusion . . . . .	66
<b>4</b>	<b>Type and Effect Discipline for C Safety</b>	<b>69</b>
4.1	Introduction . . . . .	69
4.2	Safety Type Annotations . . . . .	70
4.2.1	An Imperative Language . . . . .	71
4.2.2	Type Annotations . . . . .	72
4.2.3	Host Annotation for Type Conversions . . . . .	77
4.3	Static Safety Checks . . . . .	79
4.3.1	Safe Pointer Dereference . . . . .	80
4.3.2	Safe Pointer Deallocation . . . . .	81
4.3.3	Safe Pointer Assignment . . . . .	82
4.3.4	Safe Type Cast . . . . .	83

4.4	Typing Rules . . . . .	85
4.4.1	Typing Rules for Program Declarations . . . . .	86
4.4.2	Typing Rules for Expressions . . . . .	89
4.4.3	Typing Rules for Statements . . . . .	91
4.5	Dealing with Aliasing . . . . .	94
4.6	Type Annotations Inference . . . . .	97
4.7	Conclusion . . . . .	104
<b>5</b>	<b>Static Detection of Runtime Errors</b>	<b>105</b>
5.1	Introduction . . . . .	105
5.2	Dynamic Semantics . . . . .	106
5.3	Consistency of Static and Dynamic Semantics . . . . .	111
5.4	Soundness of Static Analysis . . . . .	128
5.5	Guiding Code Instrumentation . . . . .	132
5.5.1	Static Analysis Limitations . . . . .	132
5.5.2	Static <i>Dunno Points</i> . . . . .	133
5.5.3	Locating Instrumentation Points . . . . .	136
5.6	Extending GCC . . . . .	138
5.7	Conclusion . . . . .	141
<b>6</b>	<b>Automatic Security Verification</b>	<b>142</b>
6.1	Introduction . . . . .	142
6.2	Approach Overview . . . . .	144

6.2.1	Tree-SSA Framework . . . . .	144
6.2.2	Moped Model-Checker . . . . .	145
6.2.3	Architecture . . . . .	146
6.3	Modeling Security Properties . . . . .	150
6.3.1	Temporal Security Properties . . . . .	150
6.3.2	Pattern-based Security Automata . . . . .	151
6.3.3	From Security Automata to Remopla . . . . .	152
6.3.4	Execution of Remopla Automata . . . . .	155
6.4	Program Model Extraction . . . . .	156
6.4.1	Variable Declarations and Function Definitions . . . . .	157
6.4.2	Function Calls and Returns . . . . .	158
6.4.3	Flow Constructs . . . . .	159
6.4.4	Assignments . . . . .	160
6.5	Dealing with Data Dependencies . . . . .	160
6.5.1	Variable Declarations . . . . .	161
6.5.2	Assignment Operations . . . . .	163
6.5.3	Function Call with Parameters . . . . .	165
6.5.4	Function Return . . . . .	166
6.6	Static Analysis to Improve Abstractions . . . . .	167
6.6.1	Call-Graph Analysis . . . . .	168
6.6.2	Alias Analysis . . . . .	169
6.7	Conclusion . . . . .	171

<b>7</b>	<b>Design, Implementation, and Experimental Results</b>	<b>173</b>
7.1	Introduction . . . . .	173
7.2	Design and Implementation . . . . .	174
7.2.1	Why GIMPLE and Moped ? . . . . .	175
7.2.2	Macro Handling . . . . .	176
7.2.3	Temporary Variables . . . . .	177
7.2.4	CERT Coding Rules . . . . .	178
7.3	Experiments in Control-Flow Mode . . . . .	180
7.3.1	Unchecked Return Values . . . . .	180
7.3.2	Memory Leak Errors . . . . .	182
7.3.3	Use of Deprecated Functions . . . . .	184
7.3.4	Unsafe Environment Variables . . . . .	188
7.3.5	Race Conditions . . . . .	192
7.3.6	Unsafe Temporary File Creation . . . . .	195
7.3.7	Unsafe Creation of chroot Jail . . . . .	198
7.4	Experiments in Data-Driven Mode . . . . .	200
7.4.1	Parameter Passing . . . . .	201
7.4.2	Variable Aliasing . . . . .	202
7.4.3	Reducing False Positives . . . . .	204
7.5	Comparison with existing tools . . . . .	205
7.6	Conclusion . . . . .	206

<b>8 Conclusion</b>	<b>208</b>
8.1 Summary . . . . .	208
8.1.1 Type and Effect Discipline for C Safety . . . . .	209
8.1.2 Automatic Verification of Security Properties . . . . .	210
8.2 Future Work . . . . .	211
<b>Bibliography</b>	<b>214</b>
<b>Appendices</b>	<b>235</b>
Appendix I: Static Analysis Utility Functions . . . . .	235
Appendix II: Soundness of Inference Algorithm and Typing Rules . . . . .	239

# List of Figures

1	Automaton for the detection of bad <code>chroot()</code> usages . . . . .	55
2	Examples to illustrate annotation update of aliased variables . . . . .	98
3	Security verification framework . . . . .	147
4	Null-checking of memory allocation functions. . . . .	150
5	Global and local variable initializations. . . . .	162
6	Assignment operations. . . . .	163
7	Function call with parameters. . . . .	165
8	Function return. . . . .	166
9	Null check automaton . . . . .	181
10	Memory leak automaton . . . . .	183
11	Deprecated functions automaton . . . . .	186
12	Environment function automaton . . . . .	189
13	Race condition automaton . . . . .	192
14	Temporary file security automaton . . . . .	195
15	Secure creation of <code>chroot</code> jail automaton . . . . .	199

# List of Tables

1	Syntax of an imperative language that captures the essence of the C language . . . . .	71
2	Type and effect annotations for memory and type safety . . . . .	73
3	From declared types to inferred types and vice versa . . . . .	76
4	Static safety check for detecting unsafe dereference . . . . .	80
5	Static safety check for detecting unsafe deallocation . . . . .	81
6	Static safety check for detecting unsafe assignments . . . . .	82
7	Static safety checks for type cast operations . . . . .	83
8	Typing rules for programs, declarations, and call sites . . . . .	87
9	Typing rules for program expressions . . . . .	89
10	Typing rules for program statements . . . . .	92
11	Computable values . . . . .	107
12	Operational semantics for expressions . . . . .	109
13	Operational semantics for statements . . . . .	110
14	Function $\mathcal{F}$ whose maximal fixed point defines the consistency relation	114
15	<i>Dunno Points</i> to guide code instrumentation . . . . .	134



16	Operator $\oplus$ for combining <i>dunno points</i> . . . . .	136
17	Experimental results illustrating the performance of our approach . . .	139
18	Remopla language constructs . . . . .	145
19	Expression and statement pattern matching . . . . .	151
20	Remopla representation of program actions. . . . .	154
21	Remopla representation of program. . . . .	157
22	Model optimization by pruning security irrelevant functions. . . . .	168
23	Model optimization by pruning security irrelevant variables. . . . .	171
24	Return value checking. . . . .	182
25	Resource leak errors . . . . .	184
26	Usage of deprecated functions . . . . .	186
27	Unsafe environment variables . . . . .	190
28	File race condition TOCTTOU . . . . .	193
29	Temporary file errors . . . . .	196
30	Unsafe call to <code>chroot()</code> . . . . .	199
31	Results of the experiments in the data-driven mode . . . . .	201

# Chapter 1

## Introduction

After more than two decades of Commercial-Off-The-Shelf (COTS) hegemony, the software market witnesses an inexorable migration towards Free and Open Source Software (FOSS). COTS software used to be considered more secure, stable and mature for corporations and organizations. However, the intrinsic limitations of COTS software such as closed source code, expensive upgrades, and a lock-in effect have emerged over time. Furthermore, FOSS has achieved a great level of maturity and growth that makes it ready to compete with COTS. FOSS is developed either by volunteers, academia, non-profit organizations, or by large firms who want to include commodity software to give a competitive advantage to their hardware products. To date, thousands of FOSS projects are carried out via Internet collaboration. Many of these FOSS products are widely available and are considered to be as mature and secure as their COTS equivalents, at a lower or no cost. FOSS is now perceived

as a viable long-term solution that deserves careful consideration because of the potential for significant cost savings, improved reliability, and support advantages over proprietary software.

The principle of open source software is that the user can freely use and modify the source code. This raises a controversial debate regarding the security consequences of this principle. Some security experts claim that open source software is more secure because of the concept of peer review or multiple eyeballs where anyone can potentially examine source code, identify security flaws, and propose security fixes. Other security specialists claim the reverse arguing that malevolent parties could also take advantage of the open source to identify software vulnerabilities with the intention to exploit them. The two standpoints are defensible, however an empirical comparison between FOSS and COTS [100] has shown that software defects and vulnerabilities are detected and fixed more rapidly in open-source projects. As such, it is necessary that security issues be addressed, in a scientific manner, for open source software that tends to increase in size and complexity. Consequently, the safety and security evaluation of source code is a very important step to build secure software. The purpose of this research is the elaboration of practical, rigorous, and efficient techniques for the security and safety evaluation of FOSS.

## 1.1 Motivations and Problem Statement

Most of the existent open source software is written in the C [75] and C++ [76] programming languages which are considered as the defacto languages for system programming [90]. Such software includes operating systems (Linux [74], FreeBSD [59]), device drivers, and is complemented with Internet servers (Apache, Sendmail, Bind), databases (MySQL), etc. C/C++ fulfill performance, flexibility, strong support, and portability requirements [87]. However, security and safety features are either absent or badly supported in C/C++ programming. Lack of type safety and memory management left at programmers' discretion are the source of many critical security vulnerabilities such as buffer overflows, format string errors, and bad type conversions. We refer to memory errors and type errors in source code as low-level safety errors. Besides, the C/C++ libraries provide programmers with functions for privilege management, file management, network management, etc. These functions are designed with the care to provide flexibility and performance features at the cost of neglecting security concerns. A misuse of these functions can lead to privilege escalation, data leaks, and denial-of-service attacks. We refer to the coding rules that should prevent these kinds of errors as high-level security rules or security properties.

### 1.1.1 Need for Automated Verification Tools

Despite the availability of many books and documents that guide programmers writing safe and secure code [1, 2, 20, 73, 129], implementation errors that have severe

security consequences still exist in source code. This is because the C/C++ library functions are inherently unsafe/unsecure and should be used with precaution. Even skilled programmers tend to inadvertently commit errors that render their code vulnerable and potentially exploitable from a security standpoint. As the number of lines grows, manual checking of security violations becomes cumbersome and error-prone for programmers. Therefore, automated techniques for vulnerability detection are very helpful for programmers in diagnosing security and safety errors for the purpose of sanitizing their code. In the literature, there is a range of error detection approaches that can be mainly classified into dynamic analysis and static analysis [8, 11, 12, 32, 94]. Dynamic analysis monitors program execution to spot errors as they occur. Precision and accuracy are its key features. However, they come at the cost of a significant performance overhead induced by the runtime monitoring. Moreover, dynamic approaches suffer from incomplete path coverage as they consider one execution path at a time. The exploration of all execution paths requires the challenging definition of a large number of test cases. On the other hand, static analysis operates on source code without program execution to predict potential runtime errors. It offers the cost-saving advantage of the early detection of software errors. As opposed to the dynamic counterpart, static analysis can perform an exhaustive path coverage and does not introduce runtime overhead. In this thesis, we consider the elaboration of static analysis techniques for the security and safety evaluation of software. Our research effort focuses, among other things, on the efficiency and the usability of the proposed techniques.

### 1.1.2 Heavy Annotation Burden on Programmers

The evolution of static analysis in the arena of security and safety verification is remarkable. It went from simplistic approaches based on syntactic pattern matching to more sophisticated semantic analysis of programs. The pioneers in this field are the Lint family of tools [54, 80, 84], that started by syntactically detecting unsafe constructs in source code. Then, they moved a step forward by taking advantage of coding comments to annotate programs with security and safety constraints in order to tackle more coding errors. ITS4 [126] and RATS [118] perform a lexical analysis that improves the Lint pattern matching by identifying the meaning of the parsed elements (variables, function arguments, function calls, etc.). Unfortunately, these tools generate a very high rate of spurious warnings. However, they have the merit of being the forerunner in using static analysis for security purposes. Other tools such as CQual [58] uses a type-based analysis to detect security violations in source code. CQual annotates standard C types with user-defined qualifiers. It mainly detects violations of secure information flow where data coming from an untrustworthy source is used in a trusted destination without being checked. Through their program annotation-based techniques, the aforementioned tools specify security properties that are intermingled with the source code. This tight relation between properties and source code allows these techniques to have a deep insight into program behaviours for the purpose of uncovering undesirable ones. Nevertheless, mixing the source code with the security properties limits the expressiveness and the diversity of security properties. The latter cannot be modularly defined and applied to different

programs. Besides, programmers are reluctant to use annotation-based techniques since the manual annotation process is effort and time consuming. Therefore, our goal is to define verification techniques that release programmers from this heavy annotation burden in order to increase and facilitate their usability during software development.

### 1.1.3 Modular Security Property Specification

The diversity of software system functionalities implies a diversity of their security requirements. Therefore, programmers should have the ability to define and customize their own system-specific security rules. MOPS [31] and MC [11] are ahead of other techniques in providing flexibility and customization features for specifying security properties. Both tools provide an automata-based language to define temporal security properties related to the sequencing of program actions. The specification of security properties is isolated from the targeted source code. Therefore, the specification of a large range of security properties is worth the effort since these properties can be uniformly reused and applied to any software. These techniques utilize sophisticated compiler representations such as control-flow graphs and call graphs allowing a semantic-based analysis. However, the availability of the aforementioned tools may be an issue. The latest version 0.9.2 of MOPS has been released in 2003. In addition, MC is now a commercial tool.

A major difference between MC and MOPS that should be highlighted is that MOPS is based on model-checking, whereas MC is a static analysis tool. In fact,

new trends in software model-checking show great promise in detecting programming errors and exploring the correctness of software [18,28,35,35,37]. It is also efficient for the specification of a wide range of system-specific security properties. The model-checking process is performed on a state transition system that captures program behaviours. The model-checker exhaustively searches the state space to verify the compliance of program behaviours with respect to the specified properties. The state explosion problem is the main issue of software model-checking [38]. The number of states grows exponentially with respect to the size of the analyzed program. This problem limits the applicability and the usability of model-checking for large software verification. Abstraction is a well-known and established technique to cope with the state explosion problem by safely reducing the size of the program model. Thus, the challenge is the generation of a concise and model-checkable abstraction of the analyzed program. We believe static analysis can be used for the generation of suitable and scalable program abstractions for model-checking approaches [42].

#### **1.1.4 Efficient Synergy for Hybrid Approaches**

The notorious position of static analysis in the arena of safety and security verification of software is irrefutable. Nevertheless, the undecidability limitations of static analysis is also a fact [82], especially with imperative programming languages. For instance in C, pointer analysis such as aliasing is statically undecidable [106]. To remedy this issue, recent research trends move towards the definition of hybrid tools that establish a synergy between static and dynamic analyses to detect safety and security



violations [93]. Among these tools, we find CCured [94], SafeC [12], FailSafe [134], and RTC [72]. Generally, their main idea consists in using type inference to insert runtime checks in source code. The lightweight static analysis performed by these tools results in many runtime checks that induce a performance overhead that range from 30% to 150%. For instance, CCured performs a flow-insensitive type analysis that does not take into account pointer aliasing that is considered as crucial for the safety analysis of pointers. As a result, many superfluous runtime checks are inserted to overcome the limitations of CCured type analysis. Moreover, most of these tools only focus on spatial memory errors, i.e., out-of-bound accesses of pointers. Errors related to the bad sequencing of memory operations such as using freed pointers and double-free of pointers are not tackled. A more sophisticated type analysis is required for these tools to prune runtime checks on operations that are statically guaranteed to be safe. Besides, the type analysis should also consider temporal memory errors that may result in undesirable system crashes.

### **1.1.5 Flexible Integration of Verification Techniques**

Despite the increasing trends of automated source code checking, verification tools are not regularly used throughout software development process. The main reason is that most of these tools are not integrated into development tools and environments. Programmers are often required to learn how to configure and use security and safety verifiers. The steep learning curves of some tools discourage programmers from using them. Therefore, we face the challenge of building automated and scalable techniques

that can be efficiently integrated into the software development process in order to uncover a wide range of software vulnerabilities. More specifically, we target the severe and insidious memory and type errors of the C language that we refer to as safety errors. Additionally, we intent to provide programmers with the appealing capability of specifying and verifying system-specific properties that we refer to as high-level security properties.

## 1.2 Objectives

The ultimate goal of this work is to elaborate efficient, rigourous, and practical techniques for the automated detection of safety and security vulnerabilities in source code. More specifically, our objectives are as follows:

- Elaborate a taxonomy of coding errors that create severe vulnerabilities in source code.
- Conduct a comprehensive and comparative study of existing techniques for safety and security evaluation of source code. From this, we should identify the advantages and the shortcomings of these techniques.
- Elaborate efficient and practical approaches based on static analysis and model-checking techniques for the automatic detection of security and safety violations.
- Design and implement a safety and security verification toolkit that incorporates these analyses.

- Demonstrate the efficiency and the usability of our techniques by conducting case studies on large scale software.

## 1.3 Contributions

To pursue our objectives and solve the aforementioned related problems, we elaborate and develop techniques for the security and safety verification of software. For the verification of low-level safety properties related to type and memory errors, we present a type and effect analysis that extends the standard C type system with safety annotations. We enrich our type system with static safety checks to detect insidious safety errors. For the verification of system-specific security properties, we present an approach that combines static analysis and model-checking. The combination aims at automating the construction of model-checkable abstractions and at allowing programmers to customize their desired security properties. In the sequel, we provide more details of the aforementioned contributions.

### 1.3.1 Type and Effect Discipline for C Safety

In this thesis, we define a novel type and effect analysis for detecting memory and type safety errors in C source code. Our formalism is based on an imperative language that captures the essence of the C language. The related contributions are the following:

- Extending the standard C type system with effect, region, and host annotations to collect safety information of the analyzed program. Effects capture memory

operations in the program such as memory allocation, memory deallocation, pointer dereferencing, and pointer arithmetic. Region annotations are used to abstract dynamically allocated memory locations and declared variable memory locations. Regions also account for aliasing information, in a sense that pointer expressions annotated with the same region are referring to the same memory location. Finally, host annotations are used to track the state (allocated, dangling, wild, etc.) and the type of memory regions.

- Defining flow-sensitive type annotations for program expressions that are allowed to change at each program point. The flow-sensitivity endows our type analysis with capabilities to handle dynamic allocation and destructive updates of imperative languages. We also give an algorithm that uses region annotations to account for aliasing information and to propagate annotation updates of an expression to all its aliases.
- Defining a set of static safety checks that rely on the defined annotations to detect unsafe pointer usages and unsafe type conversions. Our static checks are compliant with the ANSI C standard [75], however they are more restrictive in order to detect coding errors that may result in runtime errors.
- Elaborating an annotation inference algorithm that automatically propagates type annotations through an intraprocedural phase and an interprocedural phase.
- Establishing the soundness proof of the inference algorithm to our type checking rules in order to use it as a decision procedure for detecting safety errors.

### 1.3.2 Static Detection of Runtime Errors

The ANSI C standard under-specifies the semantics of certain operations to allow some flexibility and portability in implementations. The under-specified behaviours of the C language can be classified into three main categories: implementation-defined behaviours, unspecified behaviours, and undefined behaviours. Implementation-defined behaviours represent details of the system that are left at implementation discretion such as the representations of integers in memory. Unspecified behaviours characterize cases for which the standard has no restriction such as the evaluation order of function arguments. Undefined behaviours that are of our interest occur when a program performs an operation that is semantically invalid and leads to system crashes. For instance, accessing uninitialized pointers, dereferencing null pointers, freeing unallocated pointers, etc. One intent of this work is to prove that our type and effect analysis is able to catch all the occurrences of its targeted type and memory errors. The related contributions are the following:

- Defining a dynamic semantics for the imperative language of our type and effect analysis. Our semantics is given in a structural operational style and is compliant with the ANSI C standard [75]. We also define a set of rules that capture runtime errors caused by our targeted set of `unsafe` type and memory operations.
- Establishing the consistency proof between the static semantics and the dynamic semantics. In other words, we prove that the values produced by evaluation of

expressions in the dynamic semantics are consistent with the types assigned to them statically.

- Establishing the soundness proof of our static analysis for the detection of the targeted type and memory errors. This proof is based on the established consistency relation between the dynamic analysis and the static analysis.
- Defining an effect-based interface that is used to supplement our static analysis with a dynamic counterpart. We use the collected effects to extract *dunno points* that characterize program points and execution paths of memory operations that are statically undecidable. These *dunno points* serve the purpose of guiding code instrumentation for dynamic analysis.

### 1.3.3 Verification of Secure Coding Rules

To target high-level security properties, we define a security verification environment that brings static analysis and model-checking into a synergy. The core idea is to utilize static analysis to generate concise and scalable abstractions of programs. Besides, programmers benefit from property expressiveness of model-checking techniques. As a result, our approach can model-check large scale software against system-specific security properties. The related contributions are the following:

- Defining a framework based on static analysis and model-checking for software security verification. The main components of the framework are the GCC compiler and the off-the-shelf model-checker Moped for pushdown systems [112].

We base our approach on the GIMPLE language-independent representation of source code provided by the GCC compiler. The intent is to define a flexible approach that can be extended to all languages that GCC supports.

- Elaborating a model generator algorithm that automatically serializes GIMPLE representation of programs into model-checkable program abstractions. The generated program abstractions prune program behaviours that are not relevant to the considered security properties. Besides, the model generator algorithm can be optionally flagged to compute data dependencies of program expressions to enhance the verification precision.
- Realizing the proposed verification approach by implementing it and conducting case studies on large scale and widely used C software. For the experiments, we verified the compliance of 35 Linux packages to a set of the CERT secure coding rules [2] that can be modeled as finite state automata.

## 1.4 Thesis Outline

The structure of this thesis is as follows. Chapter 2 gives a representative set of common coding errors and the different analyses and techniques used to reveal their presence in source code. Chapter 3 surveys the most prominent tools based on static analysis for the verification of C programs. The chapter also presents a comparative study of the techniques used by these tools while presenting their advantages and their limitations. Chapter 4 is dedicated to our type and effect discipline for memory

and type safety of C programs. The soundness of our static analysis for detecting our targeted type and runtime errors is established in Chapter 5. Chapter 6 describes our security verification framework based on static analysis and model-checking. The validation of our tool is done through the experiments detailed in Chapter 7. We draw conclusions and discuss future work in Chapter 8.



# Chapter 2

## Survey of Security and Safety

### Analyses

#### 2.1 Introduction

In this chapter, we survey coding errors and the different analyses used for their detection in source code. We focus on the C programming language [75] considered as the defacto language for system programming [91]. The standard C library of functions provide programmers with appealing capabilities over memory management, file management, privilege management, etc. These functions are designed with weak or inexistent security features. A secure usage of these functions is left to programmers' responsibility. Unfortunately, inadvertent programmers often commit coding errors that may cause security vulnerabilities. Automated verification techniques are required to assist programmers in detecting coding errors and building secure code.

The current survey chapter is organized as follows: In Section 2.2, we examine a representative set of coding errors that lead to security vulnerabilities. Section 2.3 presents the prominent analyses used to uncover these errors in source code: static analysis, dynamic analysis, and model-checking. A particular emphasis is put on static analysis as it is the cornerstone of this thesis. In Section 2.4, we compare static analysis with dynamic analysis and model-checking in order to establish their duality and their synergy.

## **2.2 What Causes Safety/Security Vulnerabilities ?**

This section gives a representative hence non exhaustive list of programming errors that can lead to critical security flaws. We classify these errors into memory management errors, race condition errors, file management errors, string manipulation errors, and privilege management errors.

### **2.2.1 Memory Management Vulnerabilities**

Memory management left at programmers discretion is an enormous source of safety and security problems for the C programming language. The programmer has complete control over allocation and deallocation of dynamic memory spaces, the size and the content of the allocated memory as well. To this end, the standard C library provides a set of functions for memory management. The inappropriate use of these functions may allow an unauthorized access to data in any memory location or to

consume all the memory locations and cause a denial of service. We present in the remainder of this section a list of common memory management errors.

### Use of Uninitialized Memory

An uninitialized or wild pointer refers to a random memory location. The dereference of a wild pointer has an undefined behavior that can lead to system crashes and memory corruptions. When the random location of a wild pointer belongs to another process, the use of that pointer results in an unauthorized memory access. To prevent such undesirable behaviors, each declared pointer must be set to null so as to refer to no memory location. The usage of a null pointer results in a segmentation fault, however it does not lead to unauthorized memory accesses. Listing 2.1 illustrates the bad usage of a wild pointer p.

Listing 2.1: Use of uninitialized pointer error

```
char *p;  
char *q = NULL;  
strcpy(p, "Hello");  
strcpy(q, "Goodbye");
```

If the random location of p is not write protected and belongs to another process, the `strcpy(p, "Hello")` call will overwrite its content and corrupt the execution of that process. On the other hand, null pointer q does not refer to any memory location. The call `strcpy(q, "Goodbye")` results in a segmentation fault.

### Use after Free

Dynamically allocated memory that is not used anymore should be explicitly released by calling the `free()` function. This good practice optimizes resource consumption

that helps increase the system performance. We refer to pointers to freed memory locations as dangling pointers. Their content may remain intact until the system decides to assign these locations to other processes. Thus, using a dangling pointer may lead to unauthorized memory access and system crashes. Listing 2.2 illustrates an unsafe usage of dangling pointer buff.

Listing 2.2: Use after free error

```
buff = (char *) malloc(BUFSIZ);
if (!buff) {
    return 0;
}
/* ... */
free(buff);
/* ... */
strncpy(buff, argv[1], BUFSIZ-1);
```

## Double Free

As detailed earlier, dynamic memory locations should be freed when no more used to speed up running processes. Inadvertent programmers often free multiple times the same memory location. The impact of these double-free errors is similar to the use of freed memory. If the freed location is assigned to another process, freeing again that location will corrupt the process execution.

Listing 2.3: Double free error

```
buff = (char *) malloc(BUFSIZ);
if (!buff) {
    return 0;
}
p = buff;
/* ... */
free(p);
/* ... */
free(buff);
```

Listing 2.3 shows an example of a double-free error. The aliasing between pointer p and pointer buff renders this kind of errors even harder to detect. There are 55

entries in the CVE database related to double free errors. In some cases, such as in MIT krb5 (CVE-2007-1216) and Mozilla Firefox (CVE-2009-0775), double free errors lead to malicious code execution on vulnerable systems.

### Free of Unallocated Memory

The `free()` function must be exclusively called on a dynamic memory location derived from `malloc`, `realloc`, and `calloc` functions. Pointers that are initialized with the address-of operator `&` refers to static memory locations on the stack. Freeing stack memory location is illegal and may lead to segmentation fault. Listing 2.4 shows examples of illegal free operations.

Listing 2.4: Illegal free operations

```
char *q;
int *p;
p = &x;
q = (char *) malloc(BUFSIZ);
q = q + 4;
/*...*/
free(p);
free(q);
```

Moreover, free operations cannot be performed on pointers that do not point to the beginning of an allocated block. In C programming, pointer arithmetic operations are used to access different elements of an array. All elements of an array have the same type, the first element is placed at the beginning of the allocated memory, and the remaining elements are placed at incremented offsets from the first one. A pointer that refers to an element inside the array cannot be freed. In Listing 2.4, the illegal free operations of pointers `q` and `p` have undefined behaviors that may result in a segmentation fault.

## Use of Unchecked Null Returns

This error often happens when programmers do not check the return value of `malloc` functions before using it. When a `malloc` function fails, it returns a null pointer. As explained earlier, using a null pointer can result in a segmentation fault. Therefore, a programmer must always check the return value of all functions that yield newly allocated pointers. In the example of Listing 2.5, pointer `p` returned by `malloc()` is not checked before being used in function `strcpy()`. There are 187 entries in CVE related to the dereference of null pointers that lead to denial of service attacks.

Listing 2.5: Unchecked null return error

```
char *p;
p = (char *) malloc(BUFSIZ);
/*...*/
strcpy(p, "Hello");
```

## Memory Leaks

A memory leak occurs when a process fails to release its assigned memory before termination. Given that operating systems assign a limited amount of memory space to each process, memory leaks degrade performance and can cause a program to run out of memory.

Listing 2.6: Memory leak error

```
char *p;
p = (char *) malloc(BUFSIZ);
if (p) {
    if (c){
        /*...*/
        return -1;
    }
    else {
        /*...*/
        free(p);
        return 0;
    }
}
```

An example of memory leak is given in Listing 10. When condition `c` evaluates to true the function returns without freeing the location of pointer `p`. Many memory leak errors listed in the CVE database are found in Linux kernel such as CVE-2009-0031, CVE-2008-2375, and CVE-2008-2136. These memory leaks consume all the kernel assigned memory space and lead to system crashes.

## Buffer Overflows

Buffer overflow errors arise from weak or non-existent bounds checking of input being stored in memory buffers [73,133]. Attacks that exploit these errors are considered as the worst security threats since they may provide attackers with a complete control over the target host. The absence of bounds checking allows attackers to overflow buffers with data that contains malicious code and overwrites the return address pointer that controls the process execution. The malicious data redirects the execution to the attacker code that will execute with the privileges of the vulnerable process. If the process is running with root privileges, the attacker will be granted full control over the victim host.

Buffer overflow errors are present in legacy code that uses deprecated functions that are readily exploitable such as `gets()`, `strcpy()`, and `strcat()`. In Listing 2.7, function `gets()` reads from the standard input and stores into `str` without performing any bound checks.

Listing 2.7: Unsafe use of deprecated functions

```
int main () {
    char *str = (char *) malloc(BUFSIZ);
    gets (str);
}
```

The standard C library provides safe replacement for these deprecated functions such as `strncpy()`, `strncat()` and `fgets()`. These functions accept a length value as a parameter which should be no larger than the size of the destination buffer. Nevertheless, a bad usage of these so-called safer functions can lead to buffer overflow errors as well. A common error in the use of the safer memory copying functions is related to wrong assumptions about the parameters. For example, `strncpy(char *s1, const char *s2, size_t n)` copies `n` bytes from `s2` to pointer `s1`. When the size `n` is not properly set, this can lead to buffer overflow even though a safer function is used as shown in Listing 2.8.

Listing 2.8: Bad handling of bounds information

```
char str1[15];
char str2[20];

strncpy(str1, str2, 20);
```

To prevent this kind of security threats, deprecated functions that suffer buffer overflow errors should never be used. Moreover, safer replacement of these functions should be used with precaution since they do not systematically eliminate buffer overflow risks. In the CVE database, there are not less than 4879 entries related to buffer overflow errors. Their harmful impact range from system crashes to remote code execution.

## 2.2.2 String Manipulation Vulnerabilities

String manipulation errors can lead to severe security breaches in programs. The C library of functions provides a set of string functions that should be used with



precaution since they suffer security pitfalls. We present in this section some coding errors related to string manipulations.

## Format String Errors

Format string errors arise from the misuse of output formatting functions that take a variable number of arguments. The C standard library contains many of these functions such as `printf()`, `fprintf()`, `sprintf()`, `snprintf()`, `vprintf()`, `vsprintf()`, and `vsnprintf()`. All these functions use a format string argument that enable programmers to specify how strings should be formatted for output. Since these functions do not entail a specific number of arguments, the C compiler cannot check for the presence of the format string argument. Besides, format strings arguments very often come from a user input that may contain a value different from what programmers expected [73]. Listing 2.9 illustrates a trivial sample of format string errors.

Listing 2.9: Unsafe format string

```
int main(int argc, char* argv[])
{
    if(argc > 1)
        printf(argv[1]);
    return 0;
}
```

In this sample, `printf()` interprets the first `"%"` character in the user defined argument `argv[1]` as the format string. For instance, if the argument contains the `"%d"` characters, the `printf()` function will use it as a format string to display a decimal integer. The function fetches for this integer in its allocated memory space called the stack, and then prints it out. This is quite dangerous since it gives attackers the possibility to read information from various memory locations. These errors can

be even more dangerous, since they may be exploited to overwrite memory locations using the "%n" format string. Generally, attackers take advantage of format string errors to inject malicious code and execute it with the privilege of a vulnerable process.

### Assumption of Null-Termination

For flexibility purposes, C programming does not entail any size limitation for strings. The C compiler depends on the presence of the null character "\0" to signal the end of strings. The idea is to keep on reading from or writing to a string until reaching the null character, thus there is no need to know in advance the length of a string. Nevertheless, if the string is not null-terminated, it can be accessed out of its boundaries and lead to buffer overflow. Unfortunately, many C string functions make the naive assumption that the last character in a string is the null character. This assumption allows for adjacent memory buffer overflows [113, 125].

Listing 2.10: Invalid null-termination assumption

```
int main() {
    char longStr[ ] = "This is a long string";
    char shortStr[16];

    strncpy(shortStr, longStr, 16);
    printf("The string is not null-terminated: %s\n",
           shortStr);
    return (0);
}
```

In Listing 2.10, the so-called safe function `strncpy()` overwrites the null character that marks the end of string `shortStr`. The subsequent `printf()` call has an undefined behavior since there is no null character that stops it from displaying output. To prevent these insidious errors, `strncpy` call should immediately be followed by the statement `shortStr[15]='\0'` to mark the end of the `shortStr` string.

### 2.2.3 Race Conditions

As stated in [27], "A serialization flaw permits the asynchronous behavior of different system components to be exploited to cause a security violation". For instance, we assume that a process *A* validates some conditions before performing a given set of operations. In the meanwhile, a second process *B* exploits a timing interval, after process *A* validates its conditions and before it performs the operations, to invalidate these conditions. Thus, process *A* will execute the set of operations with the invalid assumption that the conditions are still valid. The timing interval exploited by process *B* is created by a serialization flaw known as a race condition. They often occur in concurrent and asynchronous execution environments where multiple threads or processes access the same resources without taking care of the execution order. The Time-Of-Check-To-Time-Of-Use vulnerabilities (TOCTTOU) in file accesses [19] are a classical form of race conditions. Listing 13 illustrates an example of TOCTTOU flaw.

Listing 2.11: TOCTTOU error

```
if (access(pathname, W_OK) == 0)
fd = open(pathname, O_WRONLY);
```

Function `access()` checks the write permissions of file `pathname`. On success, function `open()` is granted the write access to the file. It is important to notice that the check operation and the access operation are not executed atomically. If the file referred by `pathname` changes between the `access()` and the `open()` calls, the executing process will access a file without checking its permissions. This is an example of TOCTTOU error in which the binding of a file name can be changed

and lead to a security flaw. In order to understand TOCTTOU binding flaws, it is important to notice the two different methods of naming objects in UNIX System [19]:

- **File path name:** it specifies the path through the file system to reach the target object. The kernel enters each folder in the path starting from the root folder until it reaches the target file. This naming method requires the kernel to traverse at least one level of indirection before reaching the target file.
- **File descriptor:** it is a unique per-process non-negative integer used for accessing open files. A file descriptor does not have any level of indirection, the kernel uses it to directly access the target file.

The naming method based on file path name is vulnerable to TOCTTOU binding flaws because of its multiple level of indirection. The kernel has to follow a long path before reaching the target object, in the meanwhile another process can change the binding to another object. The file descriptor naming method gives a direct access to the target object, thus it is much more difficult to exploit a TOCTTOU flaw when system calls use file descriptors.

#### **2.2.4 File Management Vulnerabilities**

The standard C library provides functions for creation, deletion, and manipulation of files and directories. Improper usage of these functions can lead to different security flaws such as sensitive data leaks, data corruptions, and privilege escalations. We discuss a set of file management errors in the following paragraphs.

## Improper File Permissions

In Unix systems, we refer to file creation permissions as file creation mask. The latter is divided into three categories: permissions of the user that owns the file, permissions of users that belong to the owner group, and permissions of all other users. When a file is created it inherits the current process mask which may be inconvenient to the system designer. The function `umask()` provided by the C standard library is used to explicitly set the desired access permissions of newly created files. Precaution is required when setting a file mask, when the mask is too loose unauthorized users may have access to the created files. While on the contrary, a too tight mask may deprive users of file accesses intended by the system designer. In the sample code of Listing 2.12, file `unsafefile` is created without setting a safe file `umask`. Unauthorized users may have access to `unsafefile`. For file `pathname`, the file creation mask `'077'` prevents all users except the owner from read, write, and execution accesses.

Listing 2.12: Unsafe `umask` setting

```
int fd = fopen(unsafefile, "w");
umask(077);
int fp = fopen(pathname, "w");
```

In addition to setting adequate file masks, files should be created in secure folders that have restricted access privileges. A good practice is to create files under folders that are exclusively accessed by their owners and system administrators.

## Unsafe Temporary Files

Temporary files are very often used by processes to store intermediate results or to speed up their computation. By default, many programs store their temporary files

in the `/tmp` directory with default access permissions. Programs can also use other directories for temporary files. In all cases, if the directory and file access permissions are incorrectly set, it is possible for the temporary file to be used as an attack vector for the system [129]. The temporary file can be accessed and sensitive data can be leaked. An attacker can also tamper with the content of the file and create denial of service attacks. Moreover, temporary files should be removed from the system when they are no longer used. The information they contain is very valuable for the creation of attack scenarios. Therefore, temporary encryption files, cookies, and other internet temporary files should be deleted on a regular basis for security purposes.

### **2.2.5 Privilege Management Vulnerabilities**

The least privilege principle entails that a running process should have privileges and capabilities that allow it to access resources that are necessary for its execution [21,47]. All accesses that are not required to perform its tasks should not be granted. The C library provide functions for privilege management. As for all C functions, misuses of these functions can lead to critical security threats. We present some privilege management errors of C programming in the following paragraphs.

#### **Dropping Privileges**

Setuid programs execute with the privileges of their owner, so ordinary users can access files and devices even if they do not have the required permissions. For instance, the `/usr/bin/passwd` program used to access the highly sensitive password file

`/etc/passwd` is owned by the privileged root user. Nevertheless, unprivileged users are granted root privileges for executing `/usr/bin/passwd` and changing their own password. Thus, it is mandatory to write secure `setuid` programs. A good security practice is to follow the principle of least privilege. In other words, a program should acquire the needed privileges to accomplish a task, then drop these privileges as soon as they are not needed anymore. Listing 2.13 illustrates a sample code where the acquired root privileges to execute `/usr/bin/passwd` are never dropped.

Listing 2.13: Unsafe `setuid` root code

```
setuid(0);
execl("/usr/bin/passwd", "passwd", username, NULL);
/*...*/
return 1;
```

Generally, unsafe `setuid` programs owned by root are triggered by many attackers since they enable them to gain full control over a vulnerable system.

## Chrooted Jail

As part of the least privilege principle, a process should be confined to a virtual file system that contains exclusively the files required for its computation. This good practice helps reducing the impact of an eventual attack that can gain control over a vulnerable process. The C library provides the `chroot()` function to confine a process into a virtual working directory and deny all accesses outside of it. As for all C functions, the usage of `chroot()` should meet some requirements to obtain the expected security effect.

Function `chroot()` executes with root privileges. It is obvious that the root privileges should be dropped after the `chroot()` call as mentioned previously. Nevertheless,

inadvertent programmers omit to do so and the process remains with root privileges inside the chrooted jail. This inadvertence opens a breach inside the confined directory that can be exploited by attackers to acquire high privileges. Moreover, the `chroot()` call creates the virtual directory but does not redirect processes to it. An explicit call to `chdir("/")` must be performed to confine processes into the chroot jail, otherwise the latter is ineffective. Listing 2.14 provides a trivial example of a bad `chroot()` function call.

Listing 2.14: Ineffective chroot jail

```
char path[] = "/usr/jail";
chroot(path);
/*...*/
/* program exists without performing chdir("/")*/
```

## 2.3 Vulnerability Detection Techniques

The detection techniques outlined in this section are categorized into static analysis, dynamic analysis, and model-checking. Existing tools and projects in each of these categories are presented in Chapter 3.

### 2.3.1 Static Analysis

Static analysis operates at compile time for error detection in source code. It is founded on program analysis theory for the safe prediction of program runtime behaviors [95]. Static analysis has so far been used for program optimization such as dead code elimination, loop optimization, and inline expansion [7]. The last decades



witnessed an inexorable expansion of static analysis in the arena of software security [13, 14, 16, 40, 52, 61, 116]. In this section, we give an overview of the main static analysis approaches for security and safety verification.

### **Abstract Interpretation (AI)**

Abstract Interpretation (AI) is a formal theory for constructing sound approximations of the semantics of programs [40]. It maps a program and its operations to an abstract domain and abstract semantic operations, respectively. The mapping is based on mathematical concepts and structures that provide fine tuning of the abstraction process. AI is mainly used for optimizing programs, verifying programs, and building static debugging tools. The underlying mathematical foundations guarantee the soundness of AI-based analysis. Nevertheless, the precision of the analysis is conditioned by the nature of the abstraction: (1) the abstraction might match exactly the actual behavior of the program, (2) the approximation might include extra behaviors as well as the actual behavior of the program, (3) the approximation might discard some important information about the actual behavior of the program. Efficient program abstraction techniques increase the analysis accuracy and precision. However, they also increase the analysis complexity and may render it unscalable with large software. In fact, AI-based analysis faces a trade-off between the efficiency of program approximation and the precision of the analysis.

Since the mathematics involved can be cumbersome and expensive to apply, there are few tool implementations supporting AI theory. PolySpace [104] and AbsInt [4]

are two commercial AI-based static analyzers that statically detect runtime errors. There are also academic tools such as Blast [68] and Saturn [8] that use predicate abstraction [15, 37], which is a special form of abstract interpretation in which the abstract domain is constructed using a given set of predicates.

## Type Systems

High-level languages use type systems in order to associate types with variables and expressions and to make the program more readable [26, 101]. Types facilitate the understanding of the program structure and the interpretation of the data it uses. A type system consists of a set of typing rules and a set of type inference rules. The first set of rules defines types and associates them to language objects and expressions. Type inference rules deduce the type of an untyped expression in a given program. To prevent runtime errors, type systems are augmented with type constraints that enforce preconditions on inference rules [120]. If all type constraints of a program are solved, the program is well-typed and is free of runtime errors. This feature is summarized by Milner's famous slogan "*well-typed programs cannot go wrong*" [92].

The mechanism of type constraint generation and solving is now used for the specification and the verification of security and safety constraints on programs, respectively [69, 79, 131]. The main approach consists of extending the type system with type qualifiers or annotations that hold security information. Then, security rules are expressed as constraints on these annotations [58]. For instance, the type-based tool

CQual [79] decorates standard C types with the qualifiers `user` and `kernel` to distinguish user space data from kernel space data, respectively. These qualifiers facilitate the specification of a security constraint stating that an untrustworthy user data cannot be used in function call where kernel data is expected.

Type systems provide an elegant algebraic representation of an analysis in terms of compositionality and formality that facilitates the specification of typing rules and constraints. They provide an extensible analysis that can be easily integrated into the compilation process rendering the analysis fast and scalable. Despite their efficiency for detecting low-level property violations, type systems are not suitable for the verification of high-level security properties related to program functionalities.

### **Flow Analysis**

Flow analysis is mainly categorized into control-flow analysis and data-flow analysis that are extensively applied for static program analysis [60].

Control-flow analysis computes a safe approximation of the order in which instructions of a program are executed. The control-flow graph is the most common representation used to model the flow relationships among program instructions [95]. Each node in the graph, referred to as a basic block, represents a sequence of consecutive instructions in which the flow of control enters at the beginning of the block and leaves at the end of it without any branching or jump instructions. Directed edges represent transfer of control between basic blocks. Each possible execution path of the program has a corresponding path from the entry to the exit node of the graph.

Control-flow analysis is mainly used in code optimization for detecting dead code, infinite loops, etc. In the domain of security analysis, control-flow is particularly appropriate for the verification of temporal security properties that dictate the execution order of security-relevant operations. For instance, a temporal property may state that *A call to `stat(f)` must not be followed by a call to `open(f)`.*

On the other hand, data-flow analysis is a collection of techniques that approximate the values of expressions along all possible execution paths of a program. Data-flow analysis algorithms are based on control-flow graphs extended with information about variables and expressions at each node of the graphs. Analysing the flow of data is more precise and leads to better code optimization than control-flow analysis. For example, data-flow analysis handles the problem of reaching definitions that determines for each use of a variable, the assignments that could have defined the value being read. Solving the reaching definitions problem is useful for detecting uses of undefined variables, performing constant propagation, and eliminating common subexpressions [95]. In addition, there are more sophisticated data-flow analyses such as points-to analysis [10, 119] and alias analysis [25] that focus on data memory locations. Points-to analysis is a technique that finds out to which storage locations a pointer can point during its life time. Alias analysis is a specific case of points-to analysis that finds out which pointers in a program refer to the same memory location. This information is significant for the detection of bad pointer usages in C/C++ programs.

To summarize, flow analysis has a proven efficiency in code optimization and coding error detection. The separation between the data and the control flow gives even more flexibility in using this approach in program analysis. However, this approach is still used in an ad-hoc way and has less theoretical results than some other formal techniques such as type-based analysis and model-checking.

### 2.3.2 Model-Checking

Model-checking is an automatic verification technique that has successful results for hardware verification [132]. Recently, many research groups work on model-checking for software verification and demonstrate it has very promising results [35,127]. There are two main approaches for software model-checking: temporal logic model-checking and behavioral model-checking. For these two approaches, the program is translated into a finite state model that can be given as input to a model-checker. The program model is an abstraction that can be computed using flow analysis, abstract interpretation, or type system frameworks. In the case of temporal logic model-checking, the property to check is specified in temporal logic [36]. The property expression is also given as input to the model checker. Then, the model checker performs an exhaustive state exploration to check if the model of the program satisfies the property. On the other hand, the behavioral model-checking approach compares the model of the implemented program against a model of its specification [78]. The main challenge of software model-checking is to define a precise model of the program and to deal with state explosion [38].

### 2.3.3 Dynamic Analysis

Dynamic analysis verifies the security and safety properties of a program at runtime [17]. In contrast to its static counterpart, dynamic analysis examines the actual behavior of programs without performing any approximations. Hence, the results derived from dynamic analysis are precise, though they only hold for the current execution of the program and cannot be generalized to all possible program executions. Testing is the most common technique for dynamic analysis. The effectiveness of the testing process depends upon the test data over which the program is executed. There is no guarantee that the selected test data would exercise all program execution paths to uncover property violations. As such, representative test data generation is for dynamic analysis what safe program approximation is for static analysis.

Dynamic analysis is done through code instrumentation to collect information on programs and to perform property checks as they run. Code instrumentation can either be performed on source code or on executable files. Both approaches have their advantages and their drawbacks. Instrumenting source code utilizes compilers to insert annotations at different locations of the program. This high-level source code instrumentation yields a program annotated with the needed runtime checks. On the other hand, when source code is not available, as for commercial software, the code instrumentation is performed on executable files [86]. We can find in the literature many tools based on dynamic analysis for security verification. Purify [66] is a well-established commercial tool for the detection of memory leaks, double-free errors, and out-of-bound accesses. Insure++ [99] is another commercial tool that

detects memory errors, type errors, and string manipulation errors. There are also academic tools for the dynamic detection of memory errors such as Valgrind [115] and DMalloc [128].

## 2.4 Comparative Study

Given the wide range of available approaches for analysing source code, choosing the suitable one depends on a set of criteria that we use in the comparative study presented in this section. First, we compare static analysis with dynamic in terms of completeness, soundness, performance, and cost of the analysis. Then, we compare static analysis with model-checking in terms of property expressiveness, soundness, completeness, and scalability of the analysis.

Before we embark on the comparative study, we need to clarify the meaning of soundness and completeness in the context of security and safety analysis. A program analysis is sound when it does not suffer false negatives, i.e, it does not miss any occurrence of the targeted errors. A program analysis is complete when it does not suffer false positives, i.e, all the detected errors are actual errors.

### 2.4.1 Static Analysis vs. Dynamic Analysis

Static analysis and dynamic analysis are the two main protagonists that compete for the safety and security verification of software. A comparative study of these two analyses is of great interest to reveal their duality and their potential synergy [93].

- **Soundness:** Dynamic analysis operates on actual program executions exercised by test data. The dynamic path coverage is restrained by the quality of test data. Some execution paths triggered by unexpected input data may not be considered during dynamic analysis. All property violations along these unexplored execution paths remain undiscovered. On the other hand, static analysis does not rely on any input data and performs an exhaustive examination of all predicted execution paths of programs. Static analysis strives for soundness by adopting a conservative and pessimistic approach that assumes worst-case scenarios for error detection.
- **Completeness:** By its conservative nature, static analysis is inherently imprecise and suffers from a high rate of false positives. Moreover, undecidability is a fact in static analysis that we face very often, especially with imperative programming language such as C, C++, and Java [82]. In other words, it is theoretically impossible to determine whether a given property holds for a program or not. For instance, pointer analysis such as aliasing is statically undecidable, it typically infers that two pointers *may* alias along an execution path [106]. On the other hand, completeness is the key feature of the dynamic counterpart. All errors reported during runtime are undoubtedly actual errors.
- **Performance and cost:** Dynamic analysis typically injects instrumentation code into analyzed programs. The execution of the instrumented code induces



significant time and resource overheads. Most of dynamic tools exhibited overhead between 30% to 150% compared to the execution of the unmodified programs [12, 61, 66]. As a consequence, instrumented programs can be used for testing and debugging purposes and are unsuitable in production environments with stringent timing and resource requirements. On the other hand, static analysis offers the cost-save advantage of early detection of software errors. A research study showed that static analysis tools can provide a 17% to 23% cost reduction for reported security errors [14].

From this comparison, we can deduce that static analysis and dynamic analysis are rather complementary than competitive. Combining static and dynamic analysis is a very appealing and promising approach that aim at enhancing the overall outcome of both analyses [49]. The synergy shall define a fair balance between soundness and completeness of the analysis. Recent research trends focus on combining static analysis and dynamic analysis [6, 12, 61]. These hybrid analyses insert runtime checks at program points where static analysis is undecidable. As a consequence, the number of errors detected by the overall approach increases compared to an approach exclusively based on static analysis. In addition, the number of runtime checks decreases compared to an approach exclusively based on dynamic analysis. Our type and effect analysis defined in Chapter 4 is designed with the capabilities to resort to dynamic analysis for the purpose of overcoming static undecidability. It uses an effect based approach defined in Chapter 5 to efficiently guide code instrumentation for runtime checking.

## 2.4.2 Static Analysis vs. Model-Checking

In what follows, we outline a set of criteria for the comparison of model-checking and static analysis. From this comparative study, we also deduce a duality between these two approaches that is appealing for the security verification of software.

- **Property expressiveness:** When using static security analysis, the properties to be checked should be embedded within the compiler itself. Otherwise, the latter is not able to detect errors when it does not know them. This required compiler awareness restricts the number and the customization of security and safety properties to check. Model-checking does not suffer this limitation; it allows programmers to define a wide range of system-specific properties to verify. The desired properties are expressed in temporal logic or as finite-state automata. Moreover, model-checking is more suitable for detecting high-level security properties than static analysis. The latter checks low-level properties that can be directly mapped to source code, whereas model-checking can verify properties that are implied by the code without being explicitly present in it [48].
- **Soundness and completeness:** Both static analysis and model-checking operate on program abstractions. Therefore, they both strive for soundness and cannot achieve completeness due to their conservative nature. However, there is a main difference between these two approaches in program modeling. Static analysis uses the compiler intermediate representations as program models, whereas

model-checking operates on a finite-state model of a program serialized into the input language of the model-checker. The main challenge of model-checking is the extraction of a finite-state model from the source code of programs. The model should be precise enough in order not to over-estimate and not to underestimate program behaviors. In fact, the precision of the analysis relies on the expressiveness of program abstractions. The effort and the time required by programmers to build a model-checkable abstraction hampers the usability and scalability of software model-checking. Recent research trends in software model-checking use static analysis to automate the building of a finite-state model of programs [23, 41]. Compiler intermediate representations of source code such as abstract syntax trees and control-flow graphs are structured in a way that facilitates the automation of their translation to a given model-checker input language.

- **Scalability:** Given the finite-state model of a program, the model-checking process performs an exhaustive search of the state space to ensure the conformance of the program behavior with the specified properties. State explosion problem is the main issue of software model-checking [38]. The number of states grows exponentially with respect to the size of the analyzed program. This problem limits the scalability and the usability of model-checking for large software verification. Abstraction is a well-known and established technique to cope with the state explosion problem by safely reducing the size of the program state space. During the program model construction, abstraction consists in retaining

program behaviors that are relevant for the desired properties and discarding those behaviors that are irrelevant [71]. Besides, there are also techniques that target the representation of states in memory to allow for an efficient and optimized exploration of a state space. For instance, explicit state model-checking technique stores states in hashtables and ensures that each state is explored at most once. Symbolic model-checking stores state in sophisticated and compact structures such as Binary-Decisions-Diagram (BDD) for the speed up of the exploration process [24].

The comparative study of static analysis and model-checking shows that these approaches can achieve better results when jointly performing the security and safety verification of software. We present in Chapter 6 a security verification environment that brings static analysis and model-checking into a synergy in order to leverage the advantages and overcome the shortcomings of both techniques. The core idea is to utilize static analysis for the automation of program abstraction processes. On the other hand, programmers have the ability to define a wide range of security properties using an automata-based specification approach. As a result, our approach can model-check large scale software against system-specific security properties.

## 2.5 Conclusion

In this chapter, we presented the most common coding errors of C programs that may lead to severe safety and security vulnerabilities. We also outlined the different

detection techniques used to automatically uncover coding errors. We mainly focused on static analysis that is the cornerstone of the thesis. We compared static analysis with dynamic analysis, then we compared static analysis with model-checking. The objective of the comparative study is to establish the synergies and the dualities between these approaches. In the following chapter, we present a list of existing tools that are mainly based on static analysis and model-checking for error detection.

## Chapter 3

# Survey of Security and Safety Tools

This chapter presents a survey of the prominent static analysis tools for vulnerability detection. The objective of this survey is to identify the techniques used by existing tools and the different capabilities they provide. We establish a comparative study of these tools based on the main characteristics of their approaches and their targeted security and safety violations.

This chapter is organized as follows: Section 3.1 presents well-established annotation-based tools. Automata-based tools are detailed in Section 3.2. Pioneer tools that combine static and dynamic analyses are outlined in Section 3.3. The proposed classification focuses on the main characteristics that distinguish each approach of the presented tools. We establish a comparative study of these tools in Section 3.4. We draw conclusion of this chapter in Section 3.5.

## 3.1 Annotation-Based Techniques

Program annotations serve the purpose of specifying behavioral invariants that the code must satisfy. There are two main kinds of program annotations: (1) Statement annotations that decorate elements such as condition statements, loop statements, and switch statements. (2) Type annotations that decorate standard types of programming languages. Both annotations can be used to enrich the semantics of programs with additional information that is useful for the detection of safety and security violations. We present hereafter prominent annotation-based tools that use the aforementioned techniques for security and safety verification of C programs: we detail their approach and their targeted vulnerabilities.

### 3.1.1 Lint Family

Statement annotation techniques utilize program comments to specify behavioral constraints and verify the conformance of the program to these constraints. To facilitate the reading and the understanding of their programs, almost 30% of the source code of Linux, FreeBSD, and OpenSolaris are comments [98]. From this observation, ascribing semantics for program comments shows to be an appealing approach to specify and verify security properties of source code.

Lint is considered as a pioneer tool that uses statement annotations for static verification of programs [80]. It performs a simple flow-sensitive type-checking analysis

that detects a basic set of programming errors: unused declarations, type inconsistencies, unreachable code, use before definition, infinite loops, ignored return values, and execution paths with no return. Annotations that Lint adds are also used to eliminate false positives issued from its conservative analysis. For instance, the `/*NOTREACHED*/` annotation is used to stop flagging unreachable code starting from a specific program point. The `/*FALLTHRU*/` annotations is used to stop complaining about the absence of break instruction after a case statement. An example of the usage of these annotations is given in Listing 3.1.

Listing 3.1: Lint statement annotations

```
switch (i) {
  case 10:
    i = 0;
  case 12:
    break;
  case 18:
    i = 0;
    /*FALLTHRU*/
  case 20:
    error("bad number");
    /*NOTREACHED*/
  case 22:
    return;
}
```

There are also annotations for functions with variable number of arguments such as `printf()` functions and `scanf()` functions. These functions are used to display output and to read input, respectively. They take a format string argument that specifies the format in which a data stream should be displayed or read. The absence or the bad usage of format string arguments can lead to critical security flaws know as format string vulnerabilities discussed in Chapter 2. The Lint annotations `/*PRINTFLIKEn*/` and `/*SCANFLIKEn*/` indicate that the  $n^{th}$  argument of functions



`printf()` and `scanf()` respectively should be considered as the format string argument. Most of the Lint error checks are now embedded into the GCC compiler rendering the use of Lint obsolete. Nevertheless, Lint served as a starting point and as a source of inspiration for many other static detection tools such as LCLint [53] and Splint [84] that we briefly describe hereafter.

In addition to the Lint checks, the LCLint tool makes more sophisticated usage of program comments to define behavioral constraints on the analyzed code. For instance, LCLint adds the comment `/*@modifies *a@*/` to disallow modification of variables other than `a`. The sample code of Listing 3.2 should flag a warning when checked with LCLint, since function `foo()` modifies argument `b` without being allowed to do so.

Listing 3.2: LCLint statement annotations

```
static void foo(int *a, int *b) /*@modifies *a@*/
{
    *a=1, *b=2;
}
int main()
{
    int p=10, q=20;
    foo(&p, &q);
    return 0;
}
```

The Splint tool uses more expressive annotations that tackle more programming errors than LCLint. Splint defines the clause `requires` to enforce function preconditions and the clause `ensures` to state function postconditions. The preconditions and the postconditions are added to C/C++ library functions to detect security violations such as buffer overflows. Listing 3.3 shows the annotated version of the C function `strcpy()`. The precondition states that the memory size of `s1` in lvalue position referred to by `maxSet(s1)` should be equal to or greater than the memory size of `s2`

in rvalue position referred to by `maxRead(s2)`. The postcondition states that pointers `s1` and `s2` should have the same size after the copy operation. Moreover, Splint supports user-defined annotations and the specification of syntactic constraints on their usage.

Listing 3.3: Splint statement annotations

```
char *strcpy (char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead(s2)
/\ result == s1@*/;
```

The Lint family tools are very popular as they are among the oldest tools for static verification of source code. Their lightweight analysis is able to detect common programming errors in C source code. Nevertheless, they produce many false positives and the manual annotation burden renders programmers reluctant to use these tools on large scale software.

### 3.1.2 CQual

Another commonly used program annotation technique augments standard types with qualifiers. The latter provide a natural and easy way for programmers to ensure strong invariants of programs. In fact, types ascribe semantics to programs that discipline their behaviors, e.g., arithmetic operations cannot be used on values of type strings. Thus, extending types with qualifiers augments even more the semantics of programs by expressing strong invariants. For instance, the `const` type qualifier of the C language declares an object to be constant and disallows any modification of its value. Type qualifiers can also be used to express security and safety invariants of

programs. The verification process of these invariants can easily be integrated with the type-checking process performed by compilers.

CQual is a type inference and constraint solving tool that detects vulnerabilities in C programs [58]. It provides a type qualifier framework that allows programmers to annotate standard C types with user-defined type qualifiers. These annotations refine on standard types by endowing them with security information. CQual expresses security properties as a set of constraints on annotated types. CQual verification process consists in inferring security type annotations to program expressions and checking constraint satisfaction on these inferred annotations. For instance, CQual has been used to detect user/kernel pointer errors where untrustworthy user space pointers are dereferenced inside the kernel space without being checked. To prevent these errors, CQual defines a security constraint based on two type qualifiers: `user` for unsafe pointers that are under user control, and `kernel` for safe pointers that are under kernel control. The security constraint entails that a `user` pointer can never be used where a `kernel` pointer is expected. This constraint implies an ordering relationship on qualifiers stating that `user` is a subordinate to `kernel`, and written `user < kernel`. CQual extends ordering relationships on qualifiers to subtyping relationships on qualified types according to the following built-in inference rules:

$$\frac{q_1 \leq q_2}{q_1 \text{ int} \leq q_2 \text{ int}}$$

$$\frac{q_1 \leq q_2}{q_1 \text{ ptr}(\tau) \leq q_2 \text{ ptr}(\tau)}$$

In these rules,  $q_1$  and  $q_2$  stand for type qualifiers and  $\text{ptr}(\tau)$  represents a pointer to a given type  $\tau$ .

Listing 3.4 illustrates the usage of type annotations to detect unsafe usage of user pointers. The C function `copy_from_user()` copies data from user space to kernel space. Pointer `from` is annotated with qualifier `$user`, whereas the annotation `$user * $kernel` of pointer `to` stands for a kernel pointer whose content is from user space. In the considered sample code, variable `m` and `c` belong to the kernel space. The call to `copy_from_user()` assigns a value copied from the untrustworthy user space to the kernel memory location of pointer `&m`. As such, CQual considers pointer `m` as unsafe since its content is from user space. It flags an error when the field `buf` of variable `m` is assigned to the kernel variable `c`.

Listing 3.4: CQual type annotations

```
unsigned long copy_from_user(void $user * $kernel to,
                             void * $user from, unsigned long n);
...
struct msg m;
char c;
copy_from_user(&m, (void*)arg, sizeof (m));
c = m.buf[0];
```

CQual carries out a flow-sensitive inference algorithm that generates and solves security constraints on these type qualifiers. If the inference algorithm ever fails the constraint solving, a security vulnerability is reported. CQual performs a sound analysis, but the conservative nature of its analysis may lead to false positives. Despite the soundness of CQual, the required manual annotation effort hampers programmers from porting their legacy code to annotated CQual code. Rigorous annotation effort measurements shows that an annotation overhead of one annotation per 50 lines of code comes at a cost of one programmer hour per thousand lines of code [56, 65]. CQual has also been used to detect format string vulnerabilities [116] and to verify the authorization hook placement in the Linux Security Modules [131].

## 3.2 Automata-Based Techniques

Automata-based tools are mainly based on flow analysis of source code. Their approach is based on the fact that many security properties are related to the execution sequence of security relevant operations (e.g., X should always happen before Y, X should never happen after Y). Such temporal security properties can be expressed as finite state automata that specify which sequences of actions are allowed. This section illustrates two prominent automata-based tools, namely MetaCompilation and MOPS.

### 3.2.1 Meta-Compilation

The MetaCompilation (MC) approach [11, 65] takes advantage of the compilation process to check violations of security properties in source code. Each property is expressed as a negation rule that specifies sequences of program statements that must not be executed. Programmers use a high-level automata language called *metal* [29] to express rule-checkers for security properties. MC rule-checker automata are classified into two categories [65]:

- *variable-specific* checkers that express properties related to a specific program variable such as "*a freed pointer cannot be dereferenced*".
- *global* checkers that express properties related to the whole program such as "*interrupts are disabled*".

The nodes of variable-specific automata represent states of the tracked variables, whereas the nodes of global checker represent global states of programs. The transitions between states are labeled with patterns that match C code statements involved in a security rule. For instance, the *metal* rule in Listing 3.5 detects bad usage of freed pointers in C code.

Listing 3.5: Metal checker of bad usage of dangling pointers

```
state decl any_pointer v;  
start: { kfree(v) } ==> v.freed  
  
v.freed: { *v } ==> v.stop  
  { err("using %s after free!", mc_identifier(v)); }  
| { kfree(v) } ==> v.stop,  
  { err("double free for %s!", mc_identifier(v)); }
```

The state `start` denotes the initial state of the checker. Pattern `v` matches any freed pointer in the program. For each of these pointers, an automaton instance is created to track its state and report errors in case of security violations. The pattern `kfree(v)` triggers a transition from the initial state `start` to the state `v.freed`, where pattern `v` stands for a freed pointer. From the state `v.freed`, two transitions are possible: (1) The first transition matches a pointer dereference pattern `v` and flags a use-after-free error. (2) The second matches a pointer deallocation pattern `kfree(v)` and flags a double-free error.

The MC approach operates in two phases, an intraprocedural phase and an interprocedural phase:

- The first intraprocedural pass applies the metal checkers to each basic block of the program control-flow graph generated by the compiler. The traversal of the graph uses a Depth-First Search (DFS) algorithm that goes along an execution

path until it ends. When the traversal reaches the end of a path it backtracks to the last branch point then resumes the traversal. Hence, all the execution paths of programs are explored.

- The second interprocedural pass handles function calls through a *refine* and *restore* approach of the checkers states. When a function call is followed, the objects that pass from the caller scope to the callee scope should retain their state. When the call returns, the objects states are restored in the caller scope. As the call site information is kept at each function call, this interprocedural analysis is context-sensitive.

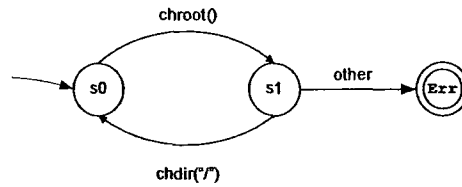
In order to reach high scalability, analysis summaries recording the states of metal checkers are kept for each analyzed function. Summaries are checked at each function call. If the recorded states of the checkers match their current states the call is not followed. The existing summaries are used to reproduce the effect of analyzing the function.

The MC approach has discovered many bugs in the Linux and BSD kernels [11] and has also been developed into a commercial product [43]. However, the scalability and the usability of MC come at the price of its analysis soundness. Many security violations such as pointer errors can pass through the checkers without being detected. These insidious errors are even more difficult to track in the presence of pointer aliasing. In fact, MC performs a trivial alias that considers simple assignment of the form  $x=y$  with no dereference operators for pointers. MC analysis also produces many

---

**Figure 1** Automaton for the detection of bad `chroot()` usages

---



---

false positives alarms, therefore the credibility of the analysis results can be affected. In order to reduce false positives, MC uses trivial data-flow analysis techniques to prune non-executable paths. Ranking heuristics are also used to sort the analysis results so that the most important security violations are displayed first in the final report [65].

### 3.2.2 MOPS

MOPS [32] is a model-checker that detects violations of temporal security properties in C programs. MOPS defines a temporal security property as a finite state automaton. Each transition in the automaton is labeled with a syntactic pattern that describes a program statement. During model-checking, a transition is taken if its label matches the current program statement. The final states of the automaton are reached when the sequence of the executed statements violates the property. For instance, the automaton shown in Figure 1 entails that the `chroot()` function call must be immediately followed by a call to `chdir("/")`.

The `chroot()` function creates an isolated virtual root directory (chrooted jail). When immediately followed by the `chdir("/")` function, the working directory is



confined to the chrooted jail. Otherwise, an attacker will still have access to files outside the chrooted jail [30].

MOPS analysis performs a pushdown model-checking algorithm to detect safety properties violations in programs [50]. MOPS model-checking algorithm operates in two steps:

- First, MOPS parses the control-flow graph of the analyzed program to derive a push-down automaton of the analyzed program [33]. Push-down automata are efficient in capturing interprocedural behaviors of programs [114]. They allow to perform an interprocedural and context-sensitive analysis since it matches each function call and function return with their corresponding call sites. It pushes the call site onto the stack at each function call and retrieves the call site from the stack at each function return.
- Second, MOPS model-checker determines whether the considered property automata reach a risky state by computing the language intersection of the property automata and the program push-down automaton. Since an automaton represents the negation of a security property, the result of the intersection should be the empty set to claim that the source code satisfies the considered properties.

MOPS has been used on the entire Red Hat Linux 9 distribution to detect file system race conditions, file descriptor vulnerabilities, and unsafe usages of temporary

files [31, 111]. MOPS model-checking is sound under the assumptions that the analyzed C program is a portable, single-threaded program that has no function pointers, signal handlers, and no runtime code generation. The pushdown-automaton model of the program enables MOPS to be context-sensitive by keeping track of the return address of each function call. On the other hand, MOPS is path-insensitive and may generate false positives related to infeasible paths. Moreover, MOPS is data-flow insensitive and does not handle aliasing neither parameter passing during program verification. For instance, we consider a TOCTTOU error that may occur when a call to `stat()` is followed by a call to `open()` on the same file name. Misled by aliasing pitfalls, MOPS cannot detect error traces such as `stat(x);y=x;open(y)`. Besides, without considering parameter passing, MOPS misses the `stat()` call followed by a subsequent call to `open()` in a different function scope as shown hereafter:

```

foo()                                int bar (char * file)
{                                     {
  /*...*/                             /*...*/
  int f = stat(f1);                    open (file);
  bar (f1);                             /*...*/
  /*...*/                               }
}

```

We show in Chapter 6 and Chapter 7 that our approach can handle variable aliasing and parameter passing for revealing insidious error traces.

### 3.3 Hybrid Techniques

Recent research trends utilize hybrid approaches that combine static and dynamic analyses for memory and type error detection [6, 12, 94]. The combination aims at

overcoming the lack of precision of static analysis by injecting runtime checks in the analyzed code. The gain in analysis precision of hybrid approaches is appealing. Nevertheless, these approaches still require an effective static analysis to reduce the number of runtime checks and their induced overhead. We discuss in this section CCured [61] and SafeC [12] considered as the pioneers of hybrid analysis tools.

### 3.3.1 CCured

CCured [61] is a hybrid tool that combines type analysis and runtime checking in order to ensure safety of C programs. It takes as input a C code and outputs an annotated CCured program instrumented with runtime checks for memory and type safety. CCured type analysis decorates the standard C type system with pointer annotations that distinguish between typed pointers and untyped pointers. The former pointers refer to objects with known types, whereas the latter refer to objects for which we cannot count on their static type due to cast operations. The annotations also define the required runtime checks to guarantee safe execution of memory operations. CCured defines the following pointer annotations:

- The `SAFE` annotation is used for pointers to typed objects. Arithmetic operations are not allowed for `SAFE` pointers, thus runtime bounds checking is not needed. The only required runtime check for `SAFE` pointers is the null-check before dereferencing.

- The SEQ annotation is used for pointers to typed objects. Arithmetic operations are allowed for SEQ pointers, thus bounds checking on these pointers is performed at runtime.
- The DYNQ annotation is used for pointers to untyped objects. The bounds and type of DYNQ pointers must be dynamically checked before dereferencing.

To support runtime bounds and type checking, CCured defines fat pointers that maintain information on their size and their type referred to as metadata. CCured modifies the code of memory allocation operations in order to initialize the metadata of a pointer when created.

CCured performs a sound analysis in the presence of runtime checks. It efficiently captures all out-of-bounds accesses and null dereferences in C programs. Nevertheless, CCured runtime checks increase drastically the performance overhead. An instrumented program may be three times slower than the original [88]. The flow-insensitive analysis and the lack of aliasing information restrain CCured from reducing the number of runtime checks. Definitely, CCured cannot be used for performance-critical software such as operating systems and drivers. Moreover, temporal memory errors such as the dereference of freed pointers and the dereference of uninitialized pointers are not detected by CCured as illustrated in Listing 3.6. The sample code `memoryErrors.c` is checked with CCured, the option `-alwaysStopOnError` generates an executable that always stops on error. However, the invocation of `memoryErrors.exe` shows that CCured does not detect nor prevent memory errors such as accessing freed

memory and use of uninitialized values.

Listing 3.6: Limitations of CCured safety analysis

```
int main(){
int * p;
int x;

p = &x;
printf("Uninit value of x:%d\n",*p);
p=malloc(sizeof(int));
printf("Uninit memory:%d\n",*p);
free(p);
*p=5;
printf("Use after free:%d\n",*p);
return 0;
}

$ ./ccured --alwaysStopOnError
memoryErrors.c -o memoryErrors.exe
$ ./memoryErrors.exe
Uninit value of x:1628796619
Uninit memory:0
Use after free:5
```

### 3.3.2 SafeC

SafeC [12] is another hybrid tool that utilizes static type inference to detect errors in source code and to insert necessary runtime checks. The main ideas of CCured approach are used in SafeC as well. SafeC changes the representation of pointers in memory in order to collect and maintain safety information such as memory bounds. Additionally, it performs a source-to-source translation for the conversion of pointer representations and the insertion of runtime checks. The augmented representation of pointers of SafeC is different from the representation of CCured fat pointers. A SafeC pointer is referred to as a safe pointer and it has the following representation:

- Field value refers to the address value referred to by the pointer.
- Fields base and size refer to the base address and the offset of the memory location, respectively.

- Field `storageClass` indicates the kind of the pointer memory region, it can be `Heap`, `Local`, or `Global`.
- Field `capability` assigns a unique identifier for dynamically allocated memory blocks.

To populate the aforementioned fields, SafeC modifies the source code of memory management functions such as the standard `malloc` functions and the `free` function. It also captures pointer creation through the `sizeof` & operator and pointer arithmetic operations.

At each pointer dereference, SafeC inserts a runtime check that refers to the base and the size pointer fields for the detection of out-of-bounds accesses. It also inserts a runtime check that uses the `storageClass` and the `capability` fields to verify that the memory region has not been deallocated. SafeC maintains a *capability store* that associates a unique identifier to each created pointer. When the pointer is freed, its identifier is removed from the store capability, thus the dereference of freed pointers can be detected. Unlike CCured, SafeC does not resort to garbage collection for the handling of freed pointers.

The static analysis of SafeC is under-exploited. It serves the purpose of inserting runtime checks and converting pointer representations. It is never used to discard runtime checks related to pointer operations that are statically guaranteed to be safe. At runtime, SafeC exhaustively checks the memory bounds and the memory state (freed or allocated) for each pointer dereference. This heavy runtime checking induces

a high performance overhead that ranges from 130% to 540 % according to the authors [12]. Moreover, the modification of the memory layout renders executables generated by SafeC incompatible with other executables generated by standard compiler such as GCC.

## 3.4 Comparative Study

From the study of existing verification tools, we distinguish a set of analysis characteristics that have an effect on the soundness, precision, scalability, and usability of these tools. We present below a listing of these characteristics in terms of their advantages and shortcomings. We also present the trade-offs that arise when choosing to design a verification tool with these listed characteristics.

### 3.4.1 Flow-Sensitive vs. Flow-Insensitive

Flow-sensitive analysis takes into account the execution order of program statements. This kind of analysis is used by MOPS, MC, and BLAST for the verification of temporal properties that entail a secure execution order of program actions. CQual also uses a flow-sensitive inference of type qualifiers that allows to set new values for qualifiers at each program point. On the other hand, flow-insensitive analysis does not consider the sequencing of program statements. Therefore, the generated results are related to the whole program and cannot be restricted to a specific program point. Flow sensitivity endows the analysis with precise results but at the cost of less

scalability. On the other hand, flow-insensitive analysis scales to large programs but do not provide accurate results. Trade-off between precision and scalability is a key factor in the choice between flow-sensitive analysis and flow-insensitive analysis.

Pointer analysis raises the dilemma of flow-sensitive or flow-insensitive analysis [70]. In order to efficiently detect memory errors, it is useful to perform a pointer analysis to know the location referred to by a pointer and the set of aliased pointers to that same location. A flow-insensitive pointer analysis is scalable but less precise [10, 119]. It can state that a given pointer  $p$  and pointer  $q$  may refer to the same location on some program paths. On the contrary, flow-sensitive approach is more precise and can derive more precise result such as pointer  $p$  and pointer  $q$  may refer to the same location at a specific program point. Nevertheless, this precision comes with the cost of less scalability [34, 130].

For instance, CCured performs a lightweight flow-insensitive pointer analysis that is not precise. To remedy the lack of precision, CCured annotates with runtime checks all pointer operations that cannot be statically proved memory safe. Nevertheless, the performance overhead introduced by the runtime checks is unacceptable and may reach 150%. Other approaches similar to CCured, such as SafeC [12], Cyclone [63], and Vault [55] have the same performance issue due to a heavy code instrumentation. This performance issue leaves us enough room to define a memory and error detection approach with a better balance between precision and effectiveness.

Yet, flow-sensitive analysis follows the control flow along the different execution paths in programs. Nevertheless, all traversed execution paths are not always actual



feasible paths at runtime. In fact, static analysis tends to be conservative and perform an exhaustive exploration of program paths without pruning infeasible ones. So, they may generate false alarms related to these impracticable paths. In this context, path-sensitivity is a desirable characteristic for increasing the precision of an analysis by discarding infeasible paths. Nevertheless, a path-sensitive approach inevitably induces sophisticated analyses that may affect its scalability to large software.

### 3.4.2 Interprocedural vs. Intraprocedural Analysis

Intraprocedural analysis considers a program function as a stand-alone entity. The results of an intraprocedural analysis do not take into account the flow of data and control at function calls, whereas interprocedural analysis does. During an interprocedural analysis, the information flows from the caller procedure to the callee procedure and vice-versa. Therefore, the results of interprocedural analysis depend on the caller-callee relation.

MC and MOPS perform an interprocedural and flow-sensitive analysis of programs. Nevertheless, the interprocedural analysis of MOPS is not efficient since it does not consider parameter passing at function boundaries as shown previously in this section. When moving from a caller function to a callee function, MOPS does not match actual parameters with formal parameters of the callee function. This prevents it from detecting errors that involve different function calls. On the other hand, the MC approach takes into account parameter passing at function boundaries. The state of a variable in a caller scope is retained and transferred to the callee

scope. This renders the interprocedural analysis of MC more precise than MOPS. The aforementioned type analysis of CCured is interprocedural but flow-insensitive. The execution order of function statements is not taken into account. The unsoundness and imprecision of CCured analysis are rectified by resorting to runtime checks that spot actual errors. A flow-sensitive analysis would help CCured reducing the number of runtime checks for performance enhancement.

A summary-based approach is often used to optimize and speed up interprocedural analysis. The idea consists in analyzing a function based on specific context and to store the analysis result in a concise summary. When the function is called again, the current call context is compared to the context of the stored summary. If they compare equal, the function is not analyzed and the stored analysis result is used. The summaries help saving time and resource consumption by avoiding the repetition of a previously performed analysis. For instance, the previously discussed MC approach achieves scalability by computing a global summary for each analyzed function that combines fine-grained summaries of each basic block of the function control-flow graph.

### **3.4.3 Internal Checking vs. External Checking**

In the review of the security tools, we have seen two kinds of checking: internal checking that uses annotations and external checking that has no annotations. Annotation-based systems specify security properties that are highly coupled with the analyzed code. Annotations are directly added to the code in order to enrich it with security

knowledge required for detecting errors. From the tight mixture between security specification and the code, annotation-based systems are more capable of achieving soundness than external checking tools. The latter provide a modular approach for security specification. The analyzed code is kept unmodified. In general, security properties are modeled by automata independently from the source code. These automata are executed during compilation or during a specific code traversal to verify security properties. Hence, the modular nature of external checking systems reduces the complexity of defining and managing security properties. However, the separation from the source code makes external checking more prone to false positives. For achieving soundness, we would state that annotated-based systems are better candidates than external checking. Nevertheless, annotated-based systems present some shortcomings that must be taken into account when designing a static checker: (1) Programmers usually do not focus on security requirements and are reluctant to specify security annotations while implementing. (2) The effort of manually adding annotations to legacy code is sometimes very hard, especially for large programs [89].

### 3.5 Conclusion

In this chapter, we presented a representative set of error detection tools mainly based on static analysis. We classified these tools into three categories that give an insight on their *modus operandi*: annotation-based techniques, automata-based techniques, and hybrid techniques. We discussed the advantages and the limitations of these

tools in detecting their targeted coding errors. From this thorough survey of error detection tools, we were able to judiciously choose the techniques and approaches that enable us to accomplish our objectives previously discussed in Chapter 1. We give in the following paragraphs a global overview of the coming chapters in which we detail our proposed methodologies for the automated security and safety verification of source code.

We target low-level coding errors related to unsafe memory operations and type conversions. The tool survey demonstrates that type-based analysis is the most suitable approach to tackle memory and types errors in C programming. As mentioned in Chapter 2, these errors are implied by the permissiveness and the flexibility of the standard C type system. Decorating the standard type system with safety annotations provides a mean to enrich the semantics of memory management and type conversions for the purpose of detecting unsafe program operations. We present in Chapter 4 our type and effect discipline for memory and type safety. The main advantage of our approach compared to the existing tools is the automated type annotation process of program expressions. Programmers are relieved from this cumbersome and heavy burden. The flow-sensitivity and alias-sensitivity features of our analysis renders it more precise than the type analysis of CCured and SafeC.

For high-level security properties related to privilege management, file management, and other system-specific properties, we advocate the usage of model-checking techniques. In the comparative study of Chapter 2, we argue that model-checking is the best candidate when it comes to property specification. It allows the definition

of customized properties tailored to specific system requirements and secure coding rules. Our security verification framework presented in Chapter 6 combines static analysis and model-checking to verify large scale C software against a set of system-specific and user-defined security rules. As opposed to MOPS, our approach computes and captures data dependencies to detect insidious errors that involve aliasing and parameter passing. Moreover, our tool provides the appealing feature of GCC multi-language support that facilitates its extension to all languages that GCC compiles

# Chapter 4

## Type and Effect Discipline for C

### Safety

#### 4.1 Introduction

This chapter presents our type and effect analysis for memory and type error detection in C code. The core idea is to decorate the standard C type system with annotations that hold safety relevant information. We also extend the standard type system with static checks that use the aforementioned annotations to detect safety violations. The flow-sensitive nature of our approach allows type annotations to change at each program statement in order to deal with the destructive updates of the imperative C language, such as dynamic allocation and deallocation of memory. Furthermore, we address the pitfalls of aliasing and indirect assignments by endowing our analysis with

flow-sensitive alias information. As such, annotation updates of a program expression are propagated to all its aliases. This chapter also details our inference algorithm that propagates type annotations to program expressions without programmers' intervention. The inference algorithm operates in two phases. The intraprocedural phase propagates type annotations and verifies memory and type operations of each function. The interprocedural phase instantiates the annotated polymorphic types of declared functions according to their actual argument types.

This chapter is organized as follows: Section 4.2 presents our imperative language that captures the essence of the C language. It also outlines the safety annotations that we decorate the standard C type system with. Our defined static checks that utilize the annotations for error detection are detailed in Section 4.3. Section 4.4 describes the typing rules for program declarations, program expressions, and program statements. Section 4.5 presents our algorithms for handling direct assignments and indirect assignments through aliasing. Section 4.6 is dedicated to our annotation inference algorithm. We conclude this chapter in Section 4.7.

## 4.2 Safety Type Annotations

In this section, we present the imperative C core that we use to illustrate our type system. We outline the annotation extensions we made to the standard C type system to ensure memory and type safety.

---

**Table 1** Syntax of an imperative language that captures the essence of the C language

---

<i>Prog</i>	$\ni \pi$	$::= \delta s$	(Program)
<i>Decl</i>	$\ni \delta$	$::= \delta_v; \delta_{fn}$	(Declarations)
<i>VarDecl</i>	$\ni \delta_v$	$::= nil$	
		$\kappa x; \delta_v$	
<i>FuncDecl</i>	$\ni \delta_{fn}$	$::= nil$	
		$\kappa' id(\kappa x) = s_{id}; \delta_{fn}$	
<i>Exp</i>	$\ni e$	$::= l_v   r_v$	(Expressions)
<i>Lval</i>	$\ni l_v$	$::= x$	(L-values)
		$*l_v$	
		$l_v.\varphi$	
<i>Rval</i>	$\ni r_v$	$::= n$	(R-values)
		$*r_v$	
		$r_v.\varphi$	
		$\&l_v$	
		$(\kappa)e$	
		$e \oplus e'$	
		$malloc(e)$	
<i>Stmnt</i>	$\ni s$	$::= free(l_v)$	(Statements)
		$l_v = e$	
		$call_{id} : l_v = id(e)$	
		<b>return</b> $e$	
		$s_1; s_2$	
		<b>if</b> $e$ <b>then</b> $s_1$ <b>else</b> $s_2$	
		<b>while</b> $e$ <b>do</b> $s$	
<i>Fields</i>	$\ni \varphi$		(Structure Fields)
<i>FuncID</i>	$\ni id$		(Function Identifiers)

---

### 4.2.1 An Imperative Language

The imperative language defined in Table 1 captures the essence of the C language [75]. A program  $\pi$  contains variable declarations  $\delta_v$  and function declarations  $\delta_{fn}$ , followed by program statements  $s$ . Without loss of generality, a function  $id$  has only one argument variable  $x$  and a body  $s_{id}$ . Program expressions comprise l-values that refer to memory locations and r-values that refer to the content of memory locations. L-values encompass variables  $x$ , l-value dereferences  $*l_v$ , and structure



fields  $l_v.\varphi$ . R-values include integer scalars  $n$ , r-value dereferences  $*r_v$ , the address of an l-value  $\&l_v$ , cast operations  $(\kappa)e$ , pointer arithmetics  $e \oplus e'$  where  $\oplus$  stands for arithmetic operators, and memory allocations  $malloc(e)$ . Statements  $s$  include memory deallocation  $free(l_v)$ , assignments  $l_v = e$ , function calls  $call_{id} : l_v = id(e)$ , return statements **return**  $e$ , and control flow constructs (sequencing, conditionals, and loops). Notice that we mark each call to function  $id$  with a label  $call_{id}$ .

### 4.2.2 Type Annotations

We present in Table 2 the type algebra of our aforementioned imperative language. In fact, our type system propagates lightweight region, effect, and host annotations that are relevant for safety analysis of C programs. They are inserted at the outermost constructor of types in order to facilitate the inference algorithm defined in Section 4.6. We present in the following paragraphs the static domains of our safety annotations.

- The domain of regions abstracts dynamic memory locations allocated on the heap and variables' memory locations assigned on the stack. The symbols  $\rho$ ,  $\rho'$  represent values drawn from this domain, a fresh symbol is derived at each memory allocation. The symbol  $\rho$  stands for a region variable with a currently unknown value. The memory location of a given variable  $x$  is given the symbolic identifier  $r_x$  where  $x$  corresponds to the unique identifier of the declared variable. Note that we use alpha-renaming to prevent collisions [81]. The notation  $\rho.o$  denotes an offset within a region  $\rho$  of a structure type. We assume that the

---

**Table 2** Type and effect annotations for memory and type safety
 

---

<i>Offset</i>	$\ni o$		
<i>Regions</i>	$\ni \rho ::=$	$\emptyset$ $\rho$ $r_x$ $\rho.o$ $\rho \cup \rho'$	
<i>Integer Host</i>	$\ni \mu ::=$	$\emptyset$ $\gamma$ $[wild]$ $[\&int]$ $[\&ref_\rho(\kappa)_\eta]$	 (uninitialized value) (initialized integer) (integer cast to pointer)
<i>Pointer Host</i>	$\ni \eta ::=$	$\mu$ $[malloc]$ $[dangling]$ $[arith]$ $[\&int_\mu]$ $[\&struct\{\_\}]$	 (allocated pointer) (freed pointer) (arithmetic pointer) (pointer to integer) (pointer to structure)
<i>Declared Types</i>	$\ni \kappa ::=$	$void$ $int$ $ref(\kappa)$ $struct\{(\varphi_i, \kappa_i)\}_{i=1..n}$ $\kappa \rightarrow \kappa'$	
<i>Inferred Types</i>	$\ni \tau ::=$	$void$ $int_\eta$ $ref_\rho(\kappa)_\eta$ $struct\{(\varphi_i, \tau_i, o_i)\}_{i=1..n}$ $\tau \xrightarrow{\sigma} \tau'$ $if(\tau, \tau')$	
<i>Effects</i>	$\ni \sigma ::=$	$\emptyset$ $\varsigma$ $alloc(\rho, \ell)$ $dealloc(\rho, \ell)$ $stack(\rho, \ell)$ $read(\rho, \tau, \ell)$ $assign(\rho, \tau, \ell)$ $arith(\rho, \ell)$ $\sigma; \sigma'$ $if(\sigma, \sigma')$	 (memory allocation) (memory deallocation) (static referencing) (memory read) (memory write) (pointer arithmetics) (sequencing of effects) (branching of effects)
<i>Program Points</i>	$\ni \ell ::=$	$l \mid 0 \mid 1 \mid 2 \mid \dots$	

---

first field is at offset 0 of the hosting location. The remaining fields are located at different offsets from the first field. As we define a flow-sensitive analysis, a pointer may refer to different regions depending on the followed branches. Hence, we use the notation  $\rho \cup \rho'$  to represent the set of disjoint regions a pointer may refer to at a given program point.

- The domain of declared types defines a representative subset of the C language types. It includes the empty type *void*, the integer type *int*, the pointer type  $ref(\kappa)$ , the structure type  $struct\{(\varphi_i, \kappa_i)\}_{i=1..n}$ , and the function type  $\kappa \rightarrow \kappa'$ .
- The domain of inferred types decorates the declared types with effect, region, and host annotations. A pointer type is annotated with the memory location  $\rho$  that it refers to. Pointer types and integer types are also annotated with pointer host annotations and integer host annotations, respectively. These host annotations capture relevant safety information related to pointer values and integer values: allocated pointer, freed pointer, uninitialized pointer, uninitialized integer, etc. Host annotations also indicate the type of values stored in the memory region of a pointers and integer variables. For converted integer and pointer types, host annotations are used to track their source type. For a pointer type  $ref_\rho(\kappa)_\eta$ , a cast operation is captured when the host annotation  $\eta$  indicates a type that is different from its declared type  $\kappa$ . For an integer type  $int_\mu$ , the annotation  $\mu$  distinguishes between an integer derived from a converted pointer and a genuine not converted integer. More details on type

conversion are given later in this section. The term  $struct\{(\varphi_i, \tau_i, o_i)\}_{i=1..n}$  is the type of a structure of  $n$  elements. Each field  $\varphi_i$  is decorated with an offset  $o_i$  from the first field at offset 0 as indicated in the ANSI C standard [75]. The function type  $\tau \xrightarrow{\sigma} \tau'$  is annotated with a latent effect  $\sigma$  that is generated when the corresponding function expression is evaluated. The conditional type construct  $if(\tau, \tau')$  denotes the type of a branching statement. Type  $\tau$  is inferred on the true branch, whereas type  $\tau'$  is inferred on the false branch. The declared types and the inferred types are related by the mean of two operators that are defined in Table 3: The operator " $\wedge$ " initially decorates declared types with host annotations set to  $[wild]$  and fresh region variables  $\rho$  for declared and yet initialized pointers. On the other hand, the operator " $\bar{\phantom{x}}$ " suppresses the annotations of inferred types and recovers their original declared types. Notice that the types in an  $if(\tau, \tau')$  construct are derived from one initial declared type. So, clearing an  $if(\tau, \tau')$  construct from its annotations is equivalent to clearing the annotations of one of its enclosed types:  $\overline{if(\tau, \tau')} = \bar{\tau} = \bar{\tau}'$

- The domain of effects captures memory operations and type conversions that are encountered at each program statement [46, 96]. We use  $\emptyset$  to denote the absence of effects, and  $\varsigma$  to denote an effect variable. Each effect records the program point  $\ell$  where it is produced. The terms  $alloc(\rho, \ell)$  and  $dealloc(\rho, \ell)$  respectively denote memory allocation and memory deallocation. The effect  $stack(\rho, \ell)$  is generated when a pointer is initialized to a stack memory location  $\rho$ . The effect  $read(\rho, \tau, \ell)$  represents the dereference of a pointer to region  $\rho$ .

---

**Table 3** From declared types to inferred types and vice versa
 

---

$$\begin{aligned}
 \widehat{void} &= void \\
 \widehat{int} &= int_{\{wild\}} \\
 \widehat{ref(\kappa)} &= ref_{\rho}(\kappa)_{\{wild\}} \quad \text{where } \rho \text{ fresh} \\
 \widehat{struct\{(\varphi_i, \kappa_i)\}} &= struct\{(\varphi_i, \hat{\kappa}_i, o_i)\}_{1..n} \\
 \widehat{\kappa \longrightarrow \kappa'} &= \hat{\kappa} \xrightarrow{\varsigma} \hat{\kappa}' \quad \text{where } \varsigma \text{ fresh}
 \end{aligned}$$

$$\begin{aligned}
 \overline{void} &= void \\
 \overline{int_{\eta}} &= int \\
 \overline{ref_{\rho}(\kappa)_{\eta}} &= ref(\kappa) \\
 \overline{struct\{(\varphi_i, \tau_i, o_i)\}_{1..n}} &= struct\{(\varphi_i, \bar{\tau}_i)\}_{1..n} \\
 \overline{\tau \xrightarrow{\sigma} \tau'} &= \bar{\tau} \longrightarrow \bar{\tau}' \\
 \overline{if(\tau, \tau')} &= \bar{\tau}
 \end{aligned}$$


---

The effect  $assign(\rho, \tau, \ell)$  represents the assignment of a value of type  $\tau$  to region  $\rho$ . The effect  $arith(\rho, \ell)$  captures pointer arithmetic operations on region  $\rho$ . Moreover, we define effects that capture control flow constructs of programs. The term  $\sigma; \sigma'$  denotes the sequencing of  $\sigma$  and  $\sigma'$ . The effect  $if(\sigma, \sigma')$  refers to a branching statement where the effects  $\sigma$  and  $\sigma'$  are respectively produced at the true branch and the false branch. Hence, the collected effects  $\sigma$  provide a tree-based model of the analyzed program that captures safety relevant operations. The static safety checks defined in Section 4.3 refer to the generated effect model in order to verify temporal properties related to the bad sequencing of memory and type operations.

### 4.2.3 Host Annotation for Type Conversions

The flexibility of the C language allows arbitrary type conversions for pointer and integer types without performing any safety checks. These explicit type casts are misleading since programmers may do wrong assumptions on the actual type of a memory location. To tackle insidious type casting errors, we refer to the host annotations of pointer and integer types to derive their actual types. As defined in Table 4.2, a pointer host annotation can be of the following values: (1) the value `[malloc]` indicates an allocated pointer, (2) the value `[dangling]` indicates a freed pointer, (3) the element `[wild]` indicates an uninitialized pointer or uninitialized integer, (4) the element `[arith]` indicates pointer arithmetics, (5) the element `[&int $\mu$ ]` represents a pointer to an integer value, and (6) the element `[&struct{__}]` stands for a region that stores a structure value. Notice that integer host annotations can refer to an integer type `[&int]` or to a pointer type `[&ref $\rho$ ( $\kappa$ ) $\eta$ ]`, since conversion between integer and pointer types is allowed. The empty set denotes the absence of host annotations and the symbol  $\gamma$  stands for a host annotation variable.

In Algorithm 1, we define two auxiliary functions that use host annotations in order to deal with type conversions:

- Function `castType( $\tau, \kappa$ )` derives an annotated type  $\tau'$  by converting  $\tau$  to an inferred type based on  $\kappa$ . As defined later in Section 4.3, we enrich our type system with static checks to uncover unsafe type conversions. For the latter, we assume that the cast operations result in the initial source type.

---

**Algorithm 1** Utility functions to deal with type conversions.
 

---

Function  $\text{castType}(\tau, \kappa) = \text{case } (\tau, \kappa) \text{ of}$

$(\text{ref}_\rho(\kappa')_\eta, \text{int})$	$\Rightarrow \text{int}_{[\&\text{ref}_\rho(\kappa')_\eta]}$
$(\text{int}_{[\&\text{ref}_\rho(\kappa')_\eta]}, \text{ref}(\kappa''))$	$\Rightarrow \text{ref}_\rho(\kappa'')_\eta$
$(\text{ref}_\rho(\text{void})_{[\text{malloc}]}, \text{ref}(\kappa''))$	$\Rightarrow \text{ref}_\rho(\kappa'')_{[\&\kappa'']}$
$(\text{ref}_\rho(\kappa')_\eta, \text{ref}(\kappa''))$	$\Rightarrow \text{ref}_\rho(\kappa'')_\eta$
$(\text{if}(\tau', \tau''), \kappa)$	$\Rightarrow \text{if}(\text{castType}(\tau', \kappa), \text{castType}(\tau'', \kappa))$
else	$\Rightarrow \tau$

end

Function  $\text{strTypeOf}(\tau) = \text{case } \tau \text{ of}$

$\text{if}(\tau', \tau'')$	$\Rightarrow \text{if}(\text{strTypeOf}(\tau'), \text{strTypeOf}(\tau''))$
$\text{ref}_\rho(\kappa)_{[\&\tau']}$	$\Rightarrow \tau'$
else	$\Rightarrow \text{void}$

end

---

- Function  $\text{strTypeOf}(\tau)$  yields the type of the value stored in the region of pointer type  $\tau$ .

We illustrate the use of these functions through the sample code of Listing 4.1.

Listing 4.1: Sample code to illustrate type casts with host annotations

```

1: typedef struct { int x,y,c; } CPnt;
2: typedef struct { int x,y; } Pnt;
3:
4: Pnt pt, *p;
5: CPnt *cp;
6: main() {
7:   p = &pt;
8:   cp = (CPnt*) p;
9:   cp->c = BLUE;
10: }
```

Initially, we infer the following types for the declared variables:

$$\begin{aligned}
 pt &\mapsto \widehat{Pnt} \text{ where } \widehat{Pnt} = \{(x, \text{int}_{[\text{wild}]}, o_1), (y, \text{int}_{[\text{wild}]}, o_2)\}. \\
 p &\mapsto \text{ref}_\rho(\widehat{Pnt})_{[\text{wild}]} \\
 cp &\mapsto \text{ref}_{\rho'}(\widehat{CPnt})_{[\text{wild}]}
 \end{aligned}$$

All host annotations are set to wild for uninitialized values, and region annotations stand for unknown values. At line (7), our analysis infers for pointer  $p$  the type  $\tau_p = \text{ref}_{r_{pt}}(\widehat{Pnt})_{\eta_p}$  where  $\eta_p = [\&\widehat{Pnt}]$ . It indicates that pointer  $p$  refers to region  $r_{pt}$

of variable  $pt$  that holds a value of type  $\widehat{Pnt}$ . At line (8), pointer  $p$  is cast from type  $\tau_p$  to type  $ref(CPnt)$ . Notice that the destination type of the conversion is defined by the programmer, thus does not have any annotation. The cast operation yields type  $\tau_{cp} = \text{castType}(\tau_p, ref(CPnt)) = ref_{\tau_{pt}}(CPnt)_{\eta_p}$ . We assume that cast operations do not change the content of memory locations; hence the host annotation of pointer  $cp$  remains  $\eta_p = [\&\widehat{Pnt}]$ . It indicates that pointer  $cp$ , initially declared to refer to a  $CPnt$  value, is actually referring to a value of type  $\widehat{Pnt}$ . With the precision of the host annotation, our analysis cannot be misled on the actual type referred to by a given pointer. Hence, we can detect that the dereference of field  $c$  at line (9) is unsafe since  $cp$  is not referring to a  $CPnt$  structure. Section 4.3 illustrates our safety checks based on host annotations for detecting type conversion errors.

### 4.3 Static Safety Checks

This section outlines the static safety checks performed by our type system to detect and prevent a set of memory and type errors that are listed in Annex J of the ANSI C standard [75]. All safety-related operations are guarded by a corresponding static check. From the conservative nature of our analysis, operations that pass the checks never cause a runtime error during program execution. Those who fail may violate memory or type safety during program execution.



---

**Table 4** Static safety check for detecting unsafe dereference

---

$$\begin{aligned} \text{drfChk}(\tau) &= (\bar{\tau} = \text{ref}(\kappa)) \wedge (\kappa \neq \text{void}) \\ &\quad \wedge (\text{hostOf}(\tau) \notin \{[\text{wild}], [\text{dangling}], [\text{arith}]\}) \\ \text{drfChk}(\text{if}(\tau, \tau')) &= \text{drfChk}(\tau) \wedge \text{drfChk}(\tau') \end{aligned}$$

---

### 4.3.1 Safe Pointer Dereference

The memory check  $\text{drfChk}(\tau)$  defined in Table 4 verifies that pointer of type  $\tau$  can be safely dereferenced. It fails in the following cases:

- Dereference of void pointers: the C language disallows dereferencing void pointers since their size and their type are unknown.
- Dereference of uninitialized pointers: an uninitialized pointer with a  $[\text{wild}]$  annotation refers to an arbitrary location that can cause harmful effects when accessed.
- Dereference of dangling pointers: a dangling pointer annotated with  $[\text{dangling}]$  keeps referring to a memory location that has already been freed. By dereferencing a dangling pointer, the original program may access memory locations that do not belong anymore to its address space, leading to undefined program behaviors.
- Dereference of arithmetic pointers: these pointers may be problematic as they may refer to out-of-bounds locations. Since we do not perform bounds checking during our type analysis, we disallow dereferencing arithmetic pointers.

---

**Table 5** Static safety check for detecting unsafe deallocation
 

---

$$\begin{aligned}
 \text{freeChk}(\tau, \sigma) &= (\bar{\tau} = \text{ref}(\kappa)) \wedge (\kappa \neq \text{void}) \\
 &\quad \wedge (\text{hostOf}(\tau) = [\&\tau']) \wedge (\text{stack}(\rho, \ell) \notin \sigma) \\
 \text{freeChk}(\text{if}(\tau, \tau'), \sigma) &= \text{freeChk}(\tau, \sigma) \wedge \text{freeChk}(\tau', \sigma)
 \end{aligned}$$


---

Notice that our static type analysis can be combined with a dynamic analysis in order to perform dynamic bound checking as presented in our work [123]. Moreover, our lightweight type annotations can be extended to carry bounds information generated by existing static bounds checking techniques [85, 105, 110].

### 4.3.2 Safe Pointer Deallocation

The static check  $\text{freeChk}(\tau, \sigma)$  detailed in Table 5 verifies that a pointer of type  $\tau$  can be safely deallocated given the collected effects  $\sigma$  of the program. It fails in the following cases:

- Deallocation of uninitialized pointers: these [*wild*] annotated pointers do not have any assigned address, and thus any attempt to free such pointers can cause undefined behaviors.
- Deallocation of dangling pointers: freeing one more time a [*dangling*] pointer may corrupt the system memory and lead to system crashes.
- Deallocation of not dynamically allocated pointers: these pointers refer to stack memory locations that have not been dynamically allocated with malloc functions, and thus cannot be dynamically freed. Notice that a pointer type

---

**Table 6** Static safety check for detecting unsafe assignments

---

$$\begin{aligned}
\text{asgnChk}(\tau, \text{int}_\eta) &= (\eta \neq [\textit{wild}]) \wedge (\textit{int} = \bar{\tau}) \\
\text{asgnChk}(\tau, \text{ref}_\rho(\kappa)_\eta) &= (\eta = [\&\tau]) \wedge (\textit{ref}(\kappa) = \bar{\tau}) \\
\text{asgnChk}(\tau, \textit{if}(\tau', \tau'')) &= \text{asgnChk}(\tau, \tau') \wedge \text{asgnChk}(\tau, \tau'')
\end{aligned}$$


---

$\text{ref}_\rho(\kappa)_{[\&\tau]}$  indicates that region  $\rho$  holds a value of type  $\tau$ . However, it does not indicate that region  $\rho$  is dynamically allocated. For that reason, we need to ensure that effect  $\textit{stack}(\rho, \ell)$  is not present in the collected effects of the program.

- Deallocation of arithmetic pointers: these  $[\textit{arith}]$  pointers may refer to out-of-bound locations, our conservative analysis disallows deallocating such pointers.

### 4.3.3 Safe Pointer Assignment

The memory check  $\text{asgnChk}(\tau, \tau')$  defined in Table 6 verifies that an l-value of type  $\tau$  can safely be assigned a right-hand-side value of type  $\tau'$ . It fails in the following cases:

- Assigning uninitialized right-hand-side value: the host annotation for integer values should be different from  $[\textit{wild}]$ . For pointer values, the host annotation should be equal to  $[\&\tau]$  indicating an initialized pointer to a value of type  $\tau$ .
- Assigning mismatched declared types: the types of the right-hand-side and the left-hand-side operators must explicitly match. As such, we avoid problematic implicit cast operations that often mislead programmers.

---

**Table 7** Static safety checks for type cast operations
 

---

<b>castChk</b> ( $\tau, \kappa$ ) = <b>case</b> ( $\tau, \kappa$ ) <b>of</b>	
$(if(\tau', \tau''), \kappa)$	$\Rightarrow$ $castChk(\tau', \kappa) \wedge castChk(\tau'', \kappa)$
$(ref_{\rho}(\kappa')_{\eta}, int)$	$\Rightarrow$ $sizeof(ref(\kappa')) \leq sizeof(int)$
$(int_{[\&\tau]}, ref(\kappa'))$	$\Rightarrow$ $(\bar{\tau}' = ref(\kappa'')) \wedge castChk(\tau', ref(\kappa'))$
$(ref_{\rho}(\kappa'')_{\eta}, ref(\kappa'))$	$\Rightarrow$ $\kappa'' \approx \kappa'$
$(ref_{\rho}(\kappa')_{\eta}, ref(void))$	$\Rightarrow$ <i>true</i>
$(ref_{\rho}(void)_{[malloc]}, ref(\kappa'))$	$\Rightarrow$ <i>true</i>
<b>else</b>	$\Rightarrow$ <i>false</i>
<b>end</b>	
$fldChk(\tau, \varphi)$	$= \varphi \in fldList(\tau)$
$fldChk(if(\tau, \tau'), \varphi)$	$= fldChk(\tau, \varphi) \wedge fldChk(\tau', \varphi)$
$fldList(struct\{(\varphi_i, \tau_i, o_i)\}_{i=1..n})$	$= [\varphi_1, \dots, \varphi_n]$
$fldList(\tau)_{\tau \neq struct\{}}$	$= \emptyset$

---

#### 4.3.4 Safe Type Cast

Explicit type casts are misleading since they make pointers refer to types that are different from their declared type. These insidious type conversions are a common source of system crashes. We use an approach to deal with type casts that is based on data memory layout and physical subtyping as defined in [117]. In Table 7, we define the static check  $castChk(\tau, \kappa)$  that takes as input the source type  $\tau$  and the destination un-annotated type  $\kappa$  of a cast operation. Notice that the destination type of a cast operation is defined by programmers and does not have any annotation. The following paragraphs outline the type cast operations considered in our analysis.

### Cast between pointers

Type conversions from a pointer type  $\tau = \text{ref}_\rho(\kappa)_\eta$  to an un-annotated pointer type  $\text{ref}(\kappa')$  are allowed provided that  $\kappa$  and  $\kappa'$  are in a physical subtyping relationship ( $\kappa \preceq \kappa'$  or  $\kappa' \preceq \kappa$ ) as defined in [117]. The physical subtyping takes into account the layouts of objects in memory. A type  $\kappa$  is considered as a physical subtype of type  $\kappa'$ , denoted  $(\kappa \preceq \kappa')$ , if memory layout of  $\kappa$  is a prefix of  $\kappa'$  memory layout. We use the notation  $\kappa \approx \kappa'$  to express that  $\kappa'$  is a subtype of  $\kappa$  or vice versa. Since our approach does not change data representation,  $\tau$  and  $\kappa = \bar{\tau}$  have the same memory layout.

### Cast from void pointers

As stated in the ANSI C standard [75], any pointer can be cast to a void pointer. A freshly allocated void pointer  $\text{ref}_\rho(\text{void})_{[\text{malloc}]}$  can always be cast to any pointer type  $\text{ref}(\kappa)$ . The conversion derives the type  $\text{ref}_\rho(\kappa)_{[\&\kappa]}$  as defined by the function `castType()` in Algorithm 1. The converted pointer  $\text{ref}_\rho(\kappa)_{[\&\kappa]}$  can be cast to any pointer type  $\text{ref}(\kappa')$  provided that  $\kappa \approx \kappa'$ .

### Cast between pointers and integers

Cast between pointers and integers is allowed provided that an integer type is large enough to hold a pointer value. However, we entail that only integers derived from pointers can be cast back to pointer type. An integer of type  $\text{int}_{[\&\text{ref}_\rho(\kappa)_\eta]}$  indicates an integer derived from pointer type  $\tau = \text{ref}_\rho(\kappa)_\eta$ . This integer can be converted to pointer type  $\text{ref}(\kappa)$  or to any pointer type  $\text{ref}(\kappa')$  where  $\kappa' \approx \kappa$ .

## Safe Field Dereference

The static check `fldChk( $\tau, \varphi$ )` verifies that a field  $\varphi$  can be safely accessed through a pointer type  $\tau$ . Due to cast operations, a pointer to a structure type  $\kappa$  can be actually referring to a structure type  $\kappa'$  where  $\kappa \approx \kappa'$ . According to the definition given in [117], a structure type  $\kappa$  is a physical subtype of structure type  $\kappa'$ , denoted as  $\kappa \preceq \kappa'$ , if the following conditions are met: (1) all the fields of  $\kappa'$  are present in  $\kappa$ , and (2) the offset of each field in  $\kappa'$  is the same in  $\kappa$ . Hence, the physical subtyping relation establishes a hierarchy for structure types. As such, a pointer of type  $\tau = \text{ref}_\rho(\kappa)_{[\&\tau']}$  can only access the common fields between the declared type  $\kappa$  and type  $\tau'$  stored in its region  $\rho$ .

## 4.4 Typing Rules

This section outlines our typing rules that are inspired from the type system for imperative languages presented in [109]. We define a type environment  $\mathcal{E}$  that maps each declared variable to an annotated type. Our type analysis infers initial annotations for declared variables that can later be updated at each program statement to capture imperative destructive updates. For declared functions, we do not enforce any restrictions on the annotations of their argument types and return type. Thus, declared functions are assigned annotation polymorphic types in environment  $\mathcal{E}$ .

Our analysis performs an intraprocedural pass and an interprocedural pass. The intraprocedural pass defines a flow-sensitive analysis that evaluates each function

body and applies the static safety checks defined in Section 4.3. The interprocedural pass instantiates the polymorphic types of callee functions by generating unification constraints at each call site [108]. The unification constraints are defined to unify pairs of region variables, host variables, program point variables, and effect variables at functions' boundaries. At each function call, our interprocedural analysis entails that the inferred function type should be equal to the declared function type, modulo type annotations. To facilitate the understanding of our typing rules, we define the following auxiliary functions:

- Function `regionOf( $\tau$ )` returns the region annotations of pointer type  $\tau$ .
- Function `addressOf( $e, \mathcal{E}$ )` returns the memory location of an l-value  $e$ . For a given pointer  $e$  of type  $\tau$ , we have `regionOf( $\tau$ ) = addressOf( $*e, \mathcal{E}$ )`.
- Function `fldType( $\tau, \varphi$ )` returns the type of field  $\varphi$  in structure type  $\tau$ .

The algorithms of these aforementioned functions are detailed in Appendix I. Through this chapter, we will write  $\mathcal{E} \dagger \mathcal{E}'$  to denote the overwriting of  $\mathcal{E}$  by  $\mathcal{E}'$ , i.e., the domain of  $\mathcal{E} \dagger \mathcal{E}'$  is  $\text{Dom}(\mathcal{E}) \cup \text{Dom}(\mathcal{E}')$ , and we have  $(\mathcal{E} \dagger \mathcal{E}')(x) = \mathcal{E}'(x)$  if  $x \in \text{Dom}(\mathcal{E}')$  and  $\mathcal{E}(x)$  otherwise.

#### 4.4.1 Typing Rules for Program Declarations

Table 8 illustrates the typing rules for programs, variable declarations, function declarations, and call sites. The sequent  $\mathcal{E}, \ell \vdash (\delta, s)$  indicates that the program containing

---

**Table 8** Typing rules for programs, declarations, and call sites
 

---

$\frac{\mathcal{E}, \ell \vdash \delta : \mathcal{E}' \quad \mathcal{E}', \ell \vdash s : \tau, \mathcal{E}'', \sigma}{\mathcal{E}, \ell \vdash (\delta, s)}$	(Program)
$\frac{}{\mathcal{E}, \ell \vdash nil : \mathcal{E}}$	(Nil-decl)
$\frac{\mathcal{E}, \ell \vdash \delta : \mathcal{E}' \quad \hat{\kappa} = \tau}{\mathcal{E}, \ell \vdash \kappa x; \delta : \mathcal{E}' \uparrow [x \mapsto \tau]}$	(Var-decl)
$\frac{\kappa_2 id(\kappa_1 x) = s_{id} \sqsubseteq \delta_{fn} \quad \tau_1 \xrightarrow{s} \tau_2 = \text{fresh}(\text{annot}(\kappa_1 \rightarrow \kappa_2))}{\bar{\tau}_1 \rightarrow \bar{\tau}_2 = \kappa_1 \rightarrow \kappa_2 \quad v_{1..n} = \text{fv}(\tau_1 \xrightarrow{s} \tau_2)}$ $\frac{}{\mathcal{E}, \ell \vdash id : \forall v_{1..n}. \tau_1 \xrightarrow{s} \tau_2}$	(Func-decl)
$\frac{\kappa_2 id(\kappa_1 x) = s_{id} \sqsubseteq \delta_{fn} \quad \mathcal{E}, \ell \vdash id : \forall v_{1..n}. \tau_1 \xrightarrow{s} \tau_2}{\tau'_1 \xrightarrow{s'} \tau'_2 = \text{fresh}(\tau_1 \xrightarrow{s} \tau_2) \quad \mathcal{E} \uparrow [x \mapsto \tau], \ell \vdash s_{id} : \tau', \mathcal{E}', \sigma}$ $\frac{\theta = \mathcal{U}(\tau'_1, \tau) \cup \mathcal{U}(\tau'_2, \tau') \cup [s' \mapsto \sigma]}{\mathcal{E}, \ell \vdash call_{id} : \tau \xrightarrow{\sigma} \tau', \mathcal{E}'}$	(Call-site)

---

declarations  $\delta$  and statements  $s$  is well-typed. We augment the initially empty environment  $\mathcal{E}$  from program declarations in  $\delta$ . The deduction  $\mathcal{E}, \ell \vdash \delta : \mathcal{E}'$  evaluates variable declarations and function declarations at program point  $\ell$ , then it yields a new environment  $\mathcal{E}'$  that is used to type-check program statements  $s$ . The judgment  $\mathcal{E}', \ell \vdash s : \tau, \mathcal{E}'', \sigma$  evaluates statement  $s$ , then yields a type  $\tau$ , an updated environment  $\mathcal{E}''$ , and a side-effect  $\sigma$  of memory operations and type conversions in statement  $s$ .

- The rule (Var-decl) maps a declared variable  $x$  of type  $\kappa$  to the annotated type  $\hat{\kappa}$  in  $\mathcal{E}$ . The operator " $\hat{\cdot}$ " sets host annotations to *wild* and region annotations of pointers to unknown regions  $\rho$  as defined in Table 3. We assume that alpha-conversion is used for renaming collision variables and avoid conflicts [81].



- The rule (Func-decl) assigns the most general type for functions in environment  $\mathcal{E}$  using fresh annotation variables. We define type schemes of the form  $\forall v_{1..n}.\tau$  where  $v_i$  can be region, effect, host, and program point annotation variables. The function `fresh()` takes the annotated type  $\text{annot}(\kappa_1 \longrightarrow \kappa_2)$  and replaces its annotation variables with fresh variables. Finally, we extend environment  $\mathcal{E}$  with a mapping from the declared function  $id$  to a polymorphic type where all free region, effect, host and program point variables in function type  $\tau_1 \xrightarrow{s} \tau_2$  are quantified. Function  $\text{fv}(\tau)$  derives the set of free variables of a given type  $\tau$ . Notice that the inferred function type should be equal to the declared type modulo type annotations. We note  $\kappa_2 id(\kappa_1 x) = s_{id} \sqsubseteq \delta_{fn}$  to indicate that there exist  $\delta'_{fn}$  and  $\delta''_{fn}$  such that  $\delta_{fn} = \delta'_{fn}; \kappa_2 id(\kappa_1 x) = s_{id}; \delta''_{fn}$
- The rule (Call-site) instantiates the type of a function  $id$  given an argument type  $\tau$ . First, it generates fresh annotations for the callee function type. Then, it unifies the argument type  $\tau$  with the generic argument type  $\tau_1$  which should be equal modulo type annotations. The unification algorithm  $\mathcal{U}$  is given in Algorithm 4 of Section 4.6. Then, the (Call-site) rule evaluates the callee function statements  $s_{id}$  and yields the following: (1) a return type  $\tau'$  that should be unifiable with the generic return type  $\tau_2$ , (2) a new type environment  $\mathcal{E}'$  that captures annotation updates at each statement, and (3) an effect  $\sigma$  that records memory operations of the callee function. Thus, judgment  $\mathcal{E}, \ell \vdash \text{call}_{id} : \tau \xrightarrow{\sigma} \tau', \mathcal{E}'$  states that when function  $id$  is invoked at program point  $\ell$  with argument type  $\tau$ , it should return a type  $\tau'$ , generate an effect  $\sigma$ , and derive a new environment  $\mathcal{E}'$ .

## 4.4.2 Typing Rules for Expressions

The sequent  $\mathcal{E}, \ell \vdash e : \tau, \sigma$  defines the typing rules for program expressions presented in Table 9. It states that under environment  $\mathcal{E}$  and at program point  $\ell$ , the evaluation of expression  $e$  returns type  $\tau$  and effect  $\sigma$ . Some of the expressions refer to critical memory and type operations. In order to ensure type and memory safety, the evaluation of these expressions is guarded by safety checks as detailed in Section 4.3.

**Table 9** Typing rules for program expressions

$\frac{\mathcal{E}(x) = \tau}{\mathcal{E}, \ell \vdash x : \tau, \emptyset}$	(Var)
$\frac{}{\mathcal{E}, \ell \vdash n : \text{int}_{[\&\text{int}]}, \emptyset}$	(Int)
$\frac{\mathcal{E}, \ell \vdash l_v : \tau, \sigma \quad \rho = \text{addressOf}(l_v, \mathcal{E})}{\mathcal{E}, \ell \vdash \&l_v : \text{ref}_{\rho}(\bar{\tau})_{[\&\tau]}, \text{stack}(\rho, \ell)}$	(Ref)
$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma \quad \bar{\tau} = \text{ref}(\_) \quad \text{drfChk}(\tau) \quad \rho = \text{regionOf}(\tau) \quad \tau' = \text{strTypeof}(\tau)}{\mathcal{E}, \ell \vdash *e : \tau', (\sigma; \text{read}(\rho, \tau', \ell))}$	(Deref)
$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma \quad \mathcal{E}, \ell \vdash e' : \text{int}_{\mu}, \sigma' \quad \bar{\tau} = \text{ref}(\_) \quad \rho = \text{regionOf}(\tau) \quad \tau' = \text{ref}_{\rho'}(\_)_{[\text{arith}]} \quad \rho' \text{ fresh}}{\mathcal{E}, \ell \vdash e \oplus e' : \tau', (\sigma; \sigma'; \text{arith}(\rho, \ell))}$	(Arith)
$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma \quad \text{castChk}(\tau, \kappa) \quad \tau' = \text{castType}(\tau, \kappa)}{\mathcal{E}, \ell \vdash \kappa(e) : \tau', \sigma}$	(Cast)
$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma \quad \bar{\tau} = \text{struct}\{\_ \} \quad \text{fldChk}(\tau, \varphi) \quad \tau' = \text{fldType}(\tau, \varphi) \quad \rho = \text{addressOf}(e, \varphi, \mathcal{E})}{\mathcal{E}, \ell \vdash e.\varphi : \tau', \sigma}$	(Field)
$\frac{}{\mathcal{E}, \ell \vdash e : \text{int}_{[\&\text{int}]}, \sigma \quad \rho \text{ fresh}}{\mathcal{E}, \ell \vdash \text{malloc}(e) : \text{ref}_{\rho}(\text{void})_{[\text{malloc}]}, (\sigma; \text{alloc}(\rho, \ell))}$	(Malloc)

- The rules (Var) and (Int) are standard rules that produce no effect. The host

annotation of a constant is set to  $[\&int]$  as it actually refers to an integer value.

- The rule (Ref) derives a pointer to the region that hosts the l-value  $l_v$ .
- The rule (Deref) dereferences a pointer expression  $e$  of type  $\tau$  and generates the effect  $read(\rho, \tau', \ell)$ , where  $\tau' = \text{strTypeOf}(\tau)$  is the actual type referred to by pointer  $e$ . Because of cast operations, type  $\tau'$  may be different from the pointer declared type. The safety of the dereference operation is guarded by the static check  $\text{drfChk}(\tau)$  detailed in Section 4.3.
- The rule (Arith) evaluates pointer arithmetic and generates the effect  $\text{arith}(\rho, \ell)$ , where  $\rho$  denotes the set of regions pointer  $e$  may refer to. We assume that a pointer arithmetic results in a fresh region  $\rho'$  with a host annotation set to  $[\text{arith}]$ .
- The rule (Cast) performs type conversion from type  $\tau$  to type  $\kappa'$ . Note that the destination type  $\kappa'$  as specified by the programmer does not have annotations. In Algorithm 1, we define function  $\text{castType}(\tau, \kappa')$  that derives an annotated type  $\tau'$  from the conversion from type  $\tau$  to type  $\kappa'$  such that  $\bar{\tau}' = \kappa'$ . In order to detect and prevent type cast errors, we define safety requirements in the static check  $\text{castChk}(\tau, \kappa')$  that should be met at each cast operation as specified in Section 4.3.
- The rule (Field) returns the type of field  $\varphi$  of a structure expression  $e$  of type  $\tau$ . The field access is guarded by the  $\text{fldChk}(\tau, \varphi)$  safety check as defined in Section 4.3.

- The rule (Malloc) returns a void pointer to a fresh region location  $\rho$  with a host annotation set to  $[malloc]$ . The allocation generates the effect  $alloc(\rho, \ell)$ . Notice that annotation  $[malloc]$  is exclusively assigned to void pointers derived from malloc functions. Once, the newly allocated pointer is cast to a given declared pointer type  $ref(\kappa)$ , the host annotation is set to  $[\&\hat{\kappa}]$ . It indicates a pointer to an uninitialized value of type  $\kappa$ .

### 4.4.3 Typing Rules for Statements

Table 10 presents the typing rules for statements. The statement judgment is of the form  $\mathcal{E}, \ell \vdash s : \tau, \mathcal{E}', \sigma$  stating that under environment  $\mathcal{E}$  and at program point  $\ell$ , the evaluation of statement  $s$  yields a type  $\tau$ , an environment  $\mathcal{E}'$ , and an effect  $\sigma$ .

The flow-sensitivity of our analysis allows us to cope with destructive updates of our imperative language by inferring new types for variables with new annotation instantiations at each program statement. As in [77], the flow-sensitivity is restricted to type annotations in order not to complicate the inference algorithm. Note that region annotations carry aliasing information in a sense that aliased pointers should have the same region annotations [121, 122]. We utilize this aliasing information to propagate annotation updates of an l-value to all its aliases that refer to the same updated region. We define in Section 4.5, function  $updEnv(\mathcal{E}, s)$  that evaluates the argument statement  $s$  under environment  $\mathcal{E}$  and yields an updated environment  $\mathcal{E}'$ .

- The rule (Free) conservatively deallocates all memory locations in  $\rho$  of pointer  $l_v$  and generates the effect  $dealloc(\rho, \ell)$ . The deallocation is guarded by the static

---

**Table 10** Typing rules for program statements
 

---

$\frac{\mathcal{E}, \ell \vdash l_v : \tau, \sigma \quad \bar{\tau} = \text{ref}(\_) \quad \text{freeChk}(\tau, \sigma) \quad \rho = \text{regionOf}(\tau) \quad \mathcal{E}' = \text{updEnv}(\mathcal{E}, \text{free}(l_v), \tau)}{\mathcal{E}, \ell \vdash \text{free}(l_v) : \text{void}, \mathcal{E}', (\sigma; \text{dealloc}(\rho, \ell))}$	(Free)
$\frac{\mathcal{E}, \ell \vdash l_v : \tau, \sigma \quad \mathcal{E}, \ell \vdash e : \tau', \sigma' \quad \text{asgnChk}(\tau, \tau') \quad \rho = \text{addressOf}(l_v, \mathcal{E}) \quad \mathcal{E}' = \text{updEnv}(\mathcal{E}, l_v = e, \tau')}{\mathcal{E}, \ell \vdash l_v = e : \tau', \mathcal{E}', (\sigma; \sigma'; \text{assign}(\rho, \tau', \ell))}$	(Assign)
$\frac{\mathcal{E}, \ell \vdash l_v : \tau, \sigma \quad \mathcal{E}, \ell \vdash \text{call}_{id} : \tau' \xrightarrow{\sigma''} \tau'', \mathcal{E}' \quad \mathcal{E}, \ell \vdash e : \tau', \sigma' \quad \rho = \text{addressOf}(l_v, \mathcal{E}) \quad \text{asgnChk}(\tau, \tau'') \quad \mathcal{E}'' = \text{updEnv}(\mathcal{E}', l_v = \text{id}(e), \tau'')}{\mathcal{E}, \ell \vdash l_v = \text{id}(e) : \tau'', \mathcal{E}'', (\sigma; \sigma'; \sigma''; \text{assign}(\rho, \tau'', \ell))}$	(Func-call)
$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma}{\mathcal{E}, \ell \vdash \text{return } e : \tau, \mathcal{E}, \sigma}$	(Func-return)
$\frac{\mathcal{E}, \ell \vdash s' : \tau', \mathcal{E}', \sigma' \quad \mathcal{E}', \ell' \vdash s'' : \tau'', \mathcal{E}'', \sigma''}{\mathcal{E}, \ell \vdash s'; s'' : \tau''; \mathcal{E}'', (\sigma'; \sigma'')}$	(Seq)
$\frac{\mathcal{E}, \ell \vdash e : \text{int}_\mu, \sigma \quad \mathcal{E}, \ell' \vdash s' : \tau', \mathcal{E}', \sigma' \quad \mathcal{E}, \ell'' \vdash s'' : \tau'', \mathcal{E}'', \sigma''}{\mathcal{E}, \ell \vdash \text{if } e \text{ then } s' \text{ else } s'' : \text{if}(\tau', \tau''), \mathcal{E}' \mathbb{M} \mathcal{E}'', (\sigma; \text{if}(\sigma', \sigma''))}$	(Cond)
$\frac{\mathcal{E}, \ell \vdash e : \text{int}_\mu, \sigma \quad \mathcal{E}, \ell' \vdash s : \tau', \mathcal{E}', \sigma'}{\mathcal{E}, \ell \vdash \text{while } e \text{ do } s : \text{if}(\tau', \text{void}), \mathcal{E}' \mathbb{M} \mathcal{E}, \text{if}((\sigma; \sigma'), \sigma)}$	(Loop)
$(\mathcal{E} \mathbb{M} \mathcal{E}')(x) = \begin{cases} \mathcal{E}(x) & \text{if } x \notin \text{Dom}(\mathcal{E}'), \\ \mathcal{E}'(x) & \text{if } x \notin \text{Dom}(\mathcal{E}), \\ \text{if}(\mathcal{E}(x), \mathcal{E}'(x)) & \text{otherwise.} \end{cases}$	

---

check  $\text{freeChk}(\tau, \sigma)$  as specified in Section 4.3. The function  $\text{updEnv}()$  yields a new environment  $\mathcal{E}'$  where host annotations of  $l_v$  and of all its aliases are set to *[dangling]*.

- The rule (Assign) assigns a value of type  $\tau'$  to an l-value  $l_v$  of type  $\tau$ . The assignment is guarded by the static check  $\text{asgnChk}(\tau, \tau')$ . The function  $\text{updEnv}()$  updates the type annotations of all variables that are directly or indirectly involved in the assignment statement. The effect  $\text{assign}(\rho, \tau', \ell)$  is generated, where  $\rho$  is the set of possible regions of the updated l-value.
- The rule (Func-call) evaluates a function call given the instantiated type of its corresponding call-site label. The return value is assigned to an l-value provided that the safety requirements of the check  $\text{asgnChk}()$  are met.
- The rule (Func-return) evaluates the return statement of the current callee function and has no effect.
- The rule (Seq) defines the sequencing of statements where the generated effect is the sequencing effect of  $s'$  and  $s''$ .
- The rule (Cond) evaluates a branching condition. We define the merge operator  $\wedge$  that assigns the type  $\text{if}(\mathcal{E}'(x), \mathcal{E}''(x))$  to a variable  $x$  at the merge point of a branching condition. It indicates that variable  $x$  is of type  $\mathcal{E}'(x)$  on the true branch and of type  $\mathcal{E}''(x)$  on the false branch. We use a similar effect construct  $\text{if}(\sigma', \sigma'')$  to denote the effect generated at the merge point of a branching condition.

- The rule (Loop) evaluates a loop construct. The resulting environment is equal to  $\mathcal{E} \mathbb{M} \mathcal{E}'$ ; it denotes that the environment remains unchanged if the loop is not entered. Otherwise, the environment  $\mathcal{E}'$  refers to the type mappings generated when the loop is entered at least one time.

## 4.5 Dealing with Aliasing

To increase the precision of our type annotation inference, we consider alias information that enables us to propagate annotation updates of an l-value to all its aliases. Pointer alias analysis has been widely investigated in years [10, 13, 25, 34, 62, 70, 83, 119, 130]. It is possible for us to integrate one of these analysis techniques as a plugin into our type system in order to get aliasing information. Nevertheless, since the region inference of our analysis carries flow-sensitive aliasing information, we advocate to use it to account for pointer aliasing in programs. This section outlines our algorithms in Algorithm 2 that handle direct and indirect assignments and update the static environment  $\mathcal{E}$  at each program point. All auxiliary functions used in these algorithms are defined in Appendix I. We give a brief description of these functions hereafter: (1) Function  $\text{updHost}(\tau, \eta)$  sets all host annotations in type  $\tau$  to  $\eta$ . (2) Function  $\text{regHostOf}(\text{ref}_\rho(\tau)_\eta)$  returns the pair  $(\rho, \eta)$  of region and host annotations of a pointer type. For conditional type  $\text{if}(\_)$  the function returns a set of pairs where each pair corresponds to one of the enclosed pointer types. (3) Function  $\text{updRegHost}(\tau, \rho, \eta)$  sets to  $\eta$  the host annotation of pointer type  $\tau$  that refers to region  $\rho$ . (4) Function

`typeOf( $e, \mathcal{E}$ )` returns the inferred type of expression  $e$  under environment  $\mathcal{E}$

- Function `updEnv( $\mathcal{E}, s, \tau$ )` in Algorithm 2 updates the current environment  $\mathcal{E}$  according to the argument statement  $s$  and the argument type  $\tau$ . It invokes function `directUpd( $\mathcal{E}, l_v, \tau$ )` defined in the following paragraph.
- Function `directUpd( $\mathcal{E}, l_v, \tau$ )` takes as arguments the current environment  $\mathcal{E}$ , the l-value  $l_v$  to be updated, and its new type  $\tau$ . After changing the annotations of the argument l-value  $l_v$ , function `aliasUpd()` updates the annotations of all aliases of  $l_v$ . Notice that modifying the annotations of a structure field implies updating the annotations of its enclosing structure type as well. Function `updFld()` handles the annotation update of aggregate types as defined in Appendix I.
- Function `aliasUpd( $\mathcal{E}, \rho, \eta$ )` takes as argument the current static environment, the updated memory location  $\rho$ , and the host annotation  $\eta$  to set to all aliased variables that refer to  $\rho$ . We illustrate in Figure 2 the different aliasing cases that we consider in our approach:
  - A variable  $x$  resides in the updated location  $\rho$  as illustrated in Sample (a) of Figure 2. The invocation `aliasUpd( $\mathcal{E}, r_x, \eta$ )` updates the host annotation of variable  $x$  in  $\mathcal{E}$ .
  - A pointer  $p$  refers to the updated location  $\rho$  with one level of indirection (one dereference operator). The aliasing information is extracted from the region annotation of pointer  $p$ , as illustrated in Sample (b) of Figure 2.



---

**Algorithm 2** Function `updEnv()` updates the static type environment at each program statement

---

```

Function updEnv( $\mathcal{E}, s, \tau$ ) =
begin
  case  $s$  of
     $free(l_v)$        $\Rightarrow$  directUpd( $\mathcal{E}, l_v, \text{updHost}(\tau, [dangling])$ )
     $call_{id} : l_v = e$   $\Rightarrow$  directUpd( $\mathcal{E}, l_v, \tau$ )
     $l_v = id(\_)$       $\Rightarrow$  directUpd( $\mathcal{E}, l_v, \tau$ )
  end
  return  $\mathcal{E}$ 
end

```

```

Function directUpd( $\mathcal{E}, l_v, \tau$ ) =
begin
  case  $l_v$  of
     $x$        $\Rightarrow$   $\mathcal{E} \dagger [x \mapsto \tau]$ 
     $x.\varphi$     $\Rightarrow$   $\mathcal{E} \dagger [x \mapsto \text{updFld}(\mathcal{E}(x), \varphi, \tau)]$ 
               aliasUpd( $\mathcal{E}, \text{addressOf}(x, \mathcal{E}), \text{hostOf}(\mathcal{E}(x))$ )
     $*l'_v.\varphi$   $\Rightarrow$   $\tau' = \text{updFld}(\text{typeOf}(*l'_v, \mathcal{E}), \varphi, \tau)$ 
               aliasUpd( $\mathcal{E}, \text{addressOf}(*l'_v, \mathcal{E}), \text{hostOf}(\tau')$ )
  end
  aliasUpd( $\mathcal{E}, \text{addressOf}(l_v, \mathcal{E}), \text{hostOf}(\tau)$ )
  return  $\mathcal{E}$ 
end

```

```

Function aliasUpd( $\mathcal{E}, \rho, \eta$ ) =
begin
  for all  $y \in \text{Dom}(\mathcal{E})$  do
    if  $\text{addressOf}(y, \mathcal{E}) \subseteq \rho$  then  $\mathcal{E} \dagger [y \mapsto \text{updHost}(\mathcal{E}(y), \eta)]$ 
    else  $\mathcal{E} \dagger [y \mapsto \text{indirectUpd}(\mathcal{E}(y), \rho, \eta)]$ 
  end
end

```

```

Function indirectUpd( $\tau, \rho, \eta$ ) =
begin let  $\tau' = \tau$  in
  for all  $(\rho', \eta') \in \text{regHostof}(\tau')$  do
    if  $\rho' \subseteq \rho$  then  $\tau' = \text{updRegHost}(\tau', \rho', \eta)$ 
    else
      if  $\eta' = [\&\tau'']$  then  $\tau' = \text{indirectUpd}(\tau'', \rho, \eta)$ 
    end
  end
  return  $\tau'$ 
end
end

```

---

The invocation `indirectUpd( $\mathcal{E}(p)$ ,  $\rho$ ,  $[\&int]$ )` updates the conditional type of pointer  $p$ . It sets the host annotation related to region  $\rho$  of pointer  $p$  to  $[\&int]$  since it has been indirectly updated through its aliased pointer  $q$ .

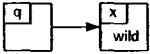
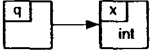
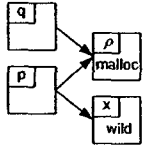
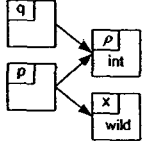
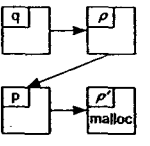
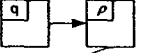
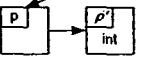
- A pointer  $p$  refers to the updated location  $\rho$  with multiple levels of indirection (more than one dereference operator). The aliasing information is extracted from the host annotation of pointer  $p$ , as shown in Sample (c) of Figure 2. The invocation `indirectUpd( $\mathcal{E}(q)$ ,  $\rho'$ ,  $[\&int]$ )` All host annotations enclosing a pointer type that refers to  $\rho'$  are updated.

## 4.6 Type Annotations Inference

This section is dedicated to the algorithm for inferring region, effect, and host annotations for program expressions. The inference algorithm proceeds by case analysis on the structure of expressions and statements. We divide the inference algorithm into three different categories: (1) annotation inference for program declarations and call-sites, (2) annotation inference for program expressions, and (3) annotation inference for program statements.

The inference algorithm for program declarations is presented in Algorithm 3. Variable declarations  $\delta_v$  and function declarations  $\delta_{fn}$  are considered separately. For the former, the algorithm takes as input a 3-tuple made of an initial static type environment  $\mathcal{E}$ , a program point  $\ell$ , and program declarations  $\delta_v$ . It evaluates the declared variables and outputs a new static environment  $\mathcal{E}'$  that maps variables to

**Figure 2** Examples to illustrate annotation update of aliased variables

<b>Sample (a):</b> Pointer $p$ refers to the memory location of $x$ .		
<pre>1: q = &amp;x;</pre>		$x: int_{wild}$ $q: ref_{rx}(int)_{\{&int_{wild}\}}$
<pre>2: *q = 5;</pre>		$x: int_{\{&int\}}$ $q: ref_{rx}(int)_{\{&int\}}$
<b>Sample (b):</b> After the branching, pointer $p$ may alias variable $x$ and pointer $q$ .		
<pre>1: q = (int *) malloc(sz); 2: if (c) 3:   p = &amp;x; 4: else 5:   p = q;</pre>		$x: int_{wild}$ $q: ref_{\rho}(int)_{\{&int_{wild}\}}$ $p: if(ref_{rx}(int)_{\{&int_{wild}\}}, ref_{\rho}(int)_{\{&int_{wild}\}})$
<pre>6: *q = 5;</pre>		$x: int_{wild}$ $q: ref_{\rho}(int)_{\{&int\}}$ $p: if(ref_{rx}(int)_{\{&int_{wild}\}}, ref_{\rho}(int)_{\{&int\}})$
<b>Sample (c):</b> The host annotations of pointer $q$ and $p$ indicate that $*q$ and $p$ are aliased		
<pre>1: q = (int *) malloc(sz); 2: p = (int **) malloc(sz);</pre>		$q: ref_{\rho}(ref(int))_{\{&ref_{\rho}(int)_{wild}\}}$ $p: ref_{\rho'}(int)_{\{&int_{wild}\}}$
<pre>3: *q = p;</pre>		$q: ref_{\rho}(ref(int))_{\{&ref_{\rho'}(int)_{\{&int_{wild}\}}\}}$
<pre>4: *p = 5;</pre>		$q: ref_{\rho}(ref(int))_{\{&ref_{\rho'}(int)_{\{&int\}}\}}$ $p: ref_{\rho'}(int)_{\{&int\}}$

---

**Algorithm 3** Annotation inference algorithm for program declarations and call-sites
 

---

```

Infer ( $\mathcal{E}, \ell, \delta_v$ ) =
  case  $\delta$  of
     $nil \Rightarrow \mathcal{E}$ 
     $\kappa x; \delta'_v \Rightarrow \text{let } \mathcal{E}' = \text{Infer}(\mathcal{E}, \ell, \delta'_v) \text{ in } \mathcal{E}' \dagger [x \mapsto \hat{\kappa}] \text{ end}$ 
  end

Infer ( $\mathcal{E}, \ell, \delta_{fn}, id$ ) =
  let  $\kappa' id(\kappa x) = s_{id} \sqsubseteq \delta_{fn}$ 
     $\tau_1 \xrightarrow{s} \tau_2 = \text{fresh}(\text{annot}(\kappa \longrightarrow \kappa'))$ 
     $v_{1..n} = \text{fv}(\tau_1 \xrightarrow{s} \tau_2)$ 
  in
     $\forall v_{1..n}. \tau_1 \xrightarrow{s} \tau_2$ 
  end

Infer ( $\mathcal{E}, \ell, \delta_{fn}, call_{id}$ ) =
  let  $\forall v_{1..n}. \tau_1 \xrightarrow{s} \tau_2 = \text{Infer}(\mathcal{E}, \ell, \delta_{fn}, id)$ 
     $(\tau', \mathcal{E}', \sigma) = \text{Infer}(\mathcal{E} \dagger [x \mapsto \tau], \ell, s_{id})$ 
     $\theta = \mathcal{U}(\tau_1, \tau) \cup \mathcal{U}(\tau_2, \tau') \cup \mathcal{U}(s, \sigma)$ 
  in
     $(\tau \xrightarrow{\sigma} \tau', \mathcal{E}')$ 
  end

```

---

annotated types with fresh region variables and host annotation set to wild. For function declarations, the algorithm takes as input a type environment  $\mathcal{E}$ , a program point  $\ell$ , function declarations  $\delta_{fn}$ , a function identifier  $id$ . It assigns for the declared function an annotation polymorphic type in  $\mathcal{E}$  with fresh annotation variables. For call sites, the algorithm instantiates the type of the callee function by defining unification constraints on type annotations. The unification algorithm  $\mathcal{U}$  defined in Algorithm 4 uses a syntactic unification procedure *à la* Robinson [108]. The proofs of soundness and completeness of  $\mathcal{U}$  are standard and can be found in [39, 108]. Notice that our analysis generates fresh annotation variables for the argument and the return types

of each function call. As such, each unification constraint generated at function boundaries is applied to fresh variables and does not override constraints related to the previous function calls. In algorithm  $\mathcal{U}$ , we use the notation  $(\eta_n)$  to denote the sequence of host annotations of a type  $\tau$  and  $(\gamma_n)$  to denote the sequence of fresh host annotation variables. In Appendix I, we define function  $\text{HostSeqOf}(\tau)$  that yields the host annotation sequence of the argument type  $\tau$ . Similarly, we write  $(\rho_n)$  to denote the sequence of region annotations of a type  $\tau$ , and  $(\varrho_n)$  denotes a sequence of fresh region annotation variables. In Appendix I, we define function  $\text{RegSeqOf}(\tau)$  that yields the region annotation sequence of the argument type  $\tau$ . As such, we have  $\text{int}_{[\eta_n]} = \tau$ , if  $\bar{\tau} = \text{int}$  and  $\text{HostSeqOf}(\tau) = [\eta_n]$ . We also have  $\text{ref}_{[\rho_n]}(\kappa)_{[\eta_n]} = \tau$ , if  $\bar{\tau} = \text{ref}(\kappa)$  and  $\text{HostSeqOf}(\tau) = [\eta_n]$  and  $\text{RegSeqOf}(\tau) = [\rho_n]$ .

---

**Algorithm 4** Syntactic unification procedure

---

$\mathcal{U}(\tau, \tau') = \text{case } (\tau, \tau') \text{ of}$ $(\text{int}_{[\eta_n]}, \text{int}_{[\eta_n]})$ $(\text{ref}_{[\varrho_n]}(\kappa)_{[\eta_n]}, \text{ref}_{[\rho_n]}(\kappa)_{[\eta_n]})$ $(\text{struct}\{(\varphi_i, \tau_i, o_i)\}, \text{struct}\{(\varphi_i, \tau'_i, o_i)\})_{i=1..n}$ $(\tau_1 \xrightarrow{\varsigma} \tau_2, \tau'_1 \xrightarrow{\sigma} \tau'_2)$ $(\text{void}, \text{void}) \mid (\text{bool}, \text{bool})$ <b>else</b> <b>end</b>	$\Rightarrow \theta = \bigcup_{i=1}^n [\gamma_i \mapsto \eta_i]$ $\Rightarrow \theta = \bigcup_{i=1}^n [\varrho_i \mapsto \rho_i] \bigcup_{i=1}^n [\gamma_i \mapsto \eta_i]$ $\Rightarrow \theta = \bigcup_{i=1}^n \mathcal{U}(\tau_i, \tau'_i)$ $\Rightarrow \theta = \mathcal{U}(\tau_1, \tau'_1) \cup \mathcal{U}(\tau_2, \tau'_2) \cup [\varsigma \mapsto \sigma]$ $\Rightarrow \text{Id}$ $\Rightarrow \text{fail}$
---	---

---

The inference algorithm for program expressions is presented in Algorithm 5. It takes as input a 3-tuple made of a static environment  $\mathcal{E}$ , a program point  $\ell$ , and an expression  $e$ . It evaluates the input expression and decorates its type with effect, region, and host annotations. When evaluating a safety-relevant expression such as

---

**Algorithm 5** Annotation inference algorithm for program expressions
 

---

**Infer** ( $\mathcal{E}, \ell, x$ ) =  
 let  $\tau = \mathcal{E}(x)$  in  $(\tau, \emptyset)$  end

**Infer** ( $\mathcal{E}, \ell, n$ ) =  $(int_{[\&int]}, \emptyset)$

**Infer** ( $\mathcal{E}, \ell, \&l_v$ ) =  
 let  $(\tau, \sigma) = \mathbf{Infer}$  ( $\mathcal{E}, \ell, l_v$ )  
 $\rho = \mathbf{addressOf}(l_v, \mathcal{E})$   
 in  
 $(ref_{\rho}(\bar{\tau})_{[\&\tau]}, (\sigma; \mathbf{stack}(\rho, \ell)))$   
 end

**Infer** ( $\mathcal{E}, \ell, *e$ ) =  
 let  $(\tau, \sigma) = \mathbf{Infer}$  ( $\mathcal{E}, \ell, e$ )  
 in  
 if  $(\mathbf{drfChk}(\tau))$  then  
 let  $\bar{\tau} = \mathbf{ref}(\_)$   
 $\tau' = \mathbf{strTypeof}(\tau)$   
 $\rho = \mathbf{regionOf}(\tau)$   
 in  
 $(\tau', (\sigma; \mathbf{read}(\rho, \tau', \ell)))$   
 else  
 fail: unsafe deref  
 end

**Infer** ( $\mathcal{E}, \ell, e \oplus e'$ ) =  
 let  $(\tau, \sigma) = \mathbf{Infer}$  ( $\mathcal{E}, \ell, e$ )  
 $\bar{\tau} = \mathbf{ref}(\_)$   
 $\rho = \mathbf{regionOf}(\tau)$   
 $(int_{\mu}, \sigma') = \mathbf{Infer}$  ( $\mathcal{E}, \ell, e'$ )  
 $\rho'$  fresh  
 in  
 $(ref_{\rho'}(\_)_{[arith]}, (\sigma; \sigma'; \mathbf{arith}(\rho, \ell)))$   
 end

**Infer** ( $\mathcal{E}, \ell, (\kappa)e$ ) =  
 let  
 $(\tau, \sigma) = \mathbf{Infer}$  ( $\mathcal{E}, \ell, e$ )  
 in  
 if  $(\mathbf{castChk}(\tau, \kappa))$  then  
 let  $\tau' = \mathbf{castType}(\tau, \kappa)$   
 in  
 $(\tau', \sigma)$   
 else  
 fail: unsafe cast  
 end

**Infer** ( $\mathcal{E}, \ell, e.\varphi$ ) =  
 let  $(\tau, \sigma) = \mathbf{Infer}$  ( $\mathcal{E}, \ell, e$ )  
 $\tau = \mathbf{struct}\{\_ \}$   
 in  
 if  $(\mathbf{fldChk}(\tau, \varphi))$  then  
 let  $\tau' = \mathbf{fldType}(\tau, \varphi)$   
 $\rho = \mathbf{addressOf}(e.\varphi, \mathcal{E})$   
 in  
 $(\tau', \sigma)$   
 else  
 fail: unsafe field access  
 end

**Infer** ( $\mathcal{E}, \ell, \mathbf{malloc}(e)$ ) =  
 let  $(\tau, \sigma) = \mathbf{Infer}$  ( $\mathcal{E}, \ell, e$ )  
 $\bar{\tau} = int_{\mu}$   
 $\rho$  fresh  
 $\tau' = ref_{\rho}(\mathbf{void})_{[\mathbf{malloc}]}$   
 in  
 $(\tau', (\sigma; \mathbf{alloc}(\rho, \ell)))$   
 end

---

**Algorithm 6** Annotation inference algorithm for program statements
 

---

```

Infer ( $\mathcal{E}, \ell, \text{free}(l_v)$ ) =
  let  $(\tau, \sigma) = \text{Infer}(\mathcal{E}, \ell, l_v)$ 
     $\bar{\tau} = \text{ref}(\_)$ 
  in
    if ( $\text{freeChk}(\tau, \sigma)$ ) then
      let  $\mathcal{E}' = \text{updEnv}(\mathcal{E}, \text{free}(l_v), \tau)$ 
         $\rho = \text{regionOf}(\tau)$ 
      in
        ( $\text{void}, \mathcal{E}', (\sigma; \text{dealloc}(\rho, \ell))$ )
      else
        fail: unsafe free
      end
    end

Infer ( $\mathcal{E}, \ell, l_v = e$ ) =
  let  $(\tau, \sigma) = \text{Infer}(\mathcal{E}, \ell, l_v)$ 
     $(\tau', \sigma') = \text{Infer}(\mathcal{E}, \ell, e)$ 
  in
    if ( $\text{asgnChk}(\tau, \tau')$ ) then
      let  $\mathcal{E}' = \text{updEnv}(\mathcal{E}, l_v = e, \tau')$ 
         $\rho = \text{addressOf}(l_v, \mathcal{E})$ 
      in
        ( $\tau', \mathcal{E}', (\sigma; \sigma'; \text{assign}(\rho, \tau', \ell))$ )
      else
        fail: unsafe assign
      end
    end

Infer ( $\mathcal{E}, \ell, \text{return } e$ ) =
  let  $(\tau, \sigma) = \text{Infer}(\mathcal{E}, \ell, e)$ 
  in
     $(\tau, \mathcal{E}, \sigma)$ 
  end

Infer ( $\mathcal{E}, \ell, l_v = \text{id}(e)$ ) =
  let  $\tau_1 \xrightarrow{\sigma''} \tau_2 = \text{Infer}(\mathcal{E}, \ell, \text{call}_{\text{id}})$ 
     $(\tau, \sigma) = \text{Infer}(\mathcal{E}, \ell, l_v)$ 
     $(\tau', \sigma') = \text{Infer}(\mathcal{E}, \ell, e)$ 
     $\text{asgnChk}(\tau, \tau_2)$ 
     $\rho = \text{addressOf}(l_v, \mathcal{E})$ 
     $\mathcal{E}'' = \text{updEnv}(\mathcal{E}', l_v = \text{id}(e), \tau_2)$ 
  in
     $(\tau_2, \mathcal{E}'', (\sigma; \sigma'; \sigma''; \text{assign}(\rho, \tau_2, \ell)))$ 
  end

Infer ( $\mathcal{E}, \ell, s'; s''$ ) =
  let  $(\tau', \mathcal{E}', \sigma') = \text{Infer}(\mathcal{E}, \ell, s')$ 
     $(\tau'', \mathcal{E}'', \sigma'') = \text{Infer}(\mathcal{E}', \ell', s'')$ 
  in
     $(\tau'', \mathcal{E}'', (\sigma'; \sigma''))$ 
  end

Infer ( $\mathcal{E}, \ell, \text{if } e \text{ then } s' \text{ else } s''$ ) =
  let  $(\text{int}_\mu, \sigma) = \text{Infer}(\mathcal{E}, \ell, e)$ 
     $(\tau', \mathcal{E}', \sigma') = \text{Infer}(\mathcal{E}, \ell', s')$ 
     $(\tau'', \mathcal{E}'', \sigma'') = \text{Infer}(\mathcal{E}, \ell'', s'')$ 
  in
     $(\text{if}(\tau', \tau''), \mathcal{E}' \mathbb{M} \mathcal{E}'', (\sigma; \text{if}(\sigma'; \sigma'')))$ 
  end

Infer ( $\mathcal{E}, \ell, \text{while } e \text{ do } s$ ) =
  let  $(\text{int}_\mu, \sigma) = \text{Infer}(\mathcal{E}, \ell, e)$ 
     $(\tau', \mathcal{E}', \sigma') = \text{Infer}(\mathcal{E}, \ell', s)$ 
  in
     $(\text{if}(\text{void}, \tau), \mathcal{E} \mathbb{M} \mathcal{E}', \text{if}(\sigma; (\sigma; \sigma')))$ 
  end

```

---

pointer dereferencing, type casts, and structure field accesses, the algorithm applies the required safety checks defined in Section 4.3. When these checks fail, the inference algorithm fails as well. The inference algorithm for program statements is presented in Algorithm 6. It takes as input a 3-tuple made of a static environment  $\mathcal{E}$ , a program point  $\ell$ , and a statement  $s$ . The algorithm fails when the safety checks related to the considered statement fail. Otherwise, the algorithm terminates successfully producing a 3-tuple enclosing an annotated type  $\tau$ , a new static environment  $\mathcal{E}$ , and an effect  $\sigma$ .

In order for the annotation inference algorithm to serve as a static detection system for memory and type errors, it must be sound with respect to the typing rules defined in Section 4.4. In other words, a typing judgment inferred by the type annotation inference algorithm must be deducible by the typing rules as stated by the Inference Algorithm Soundness 4.6.1.

**Theorem 4.6.1 (Inference Soundness)** *Given a type environment  $\mathcal{E}$ , a program point  $\ell$ , variable declarations  $\delta_v$ , function declarations  $\delta_{fn}$ , a call site  $call_{id}$ , an expression  $e$ , and a statement  $s$ , we have:*

- *If  $\text{Infer}(\mathcal{E}, \ell, \delta_v) = \mathcal{E}'$ , then  $\mathcal{E}, \ell \vdash \delta : \mathcal{E}'$*
- *If  $\text{Infer}(\mathcal{E}, \ell, \delta_{fn}, id) = \forall v_{1..n}. \tau$ , then  $\mathcal{E}, \ell \vdash id : \forall v_{1..n}. \tau$*
- *If  $\text{Infer}(\mathcal{E}, \ell, call_{id}) = (\tau, \mathcal{E}')$ , then  $\mathcal{E}, \ell \vdash call_{id} : \tau, \mathcal{E}'$*
- *If  $\text{Infer}(\mathcal{E}, \ell, e) = (\tau, \sigma)$ , then  $\mathcal{E}, \ell \vdash e : \tau, \sigma$*
- *If  $\text{Infer}(\mathcal{E}, \ell, s) = (\tau, \mathcal{E}', \sigma)$ , then  $\mathcal{E}, \ell \vdash s : \tau, \mathcal{E}', \sigma$*



**Proof of Inference Soundness Theorem 4.6.1** In Appendix II, we establish this desired error detection property by proving the Soundness Theorem. ■

## 4.7 Conclusion

In this chapter, we presented a type and effect discipline for detecting memory and type errors in C source code. Our type analysis propagates effect, region, and host annotations that carry safety knowledge regarding the analyzed program. We endow our type system with static safety checks that use the aforementioned annotations to uncover memory and type errors. Our safety analysis performs in an intraprocedural phase and an interprocedural phase: (1) The intraprocedural phase infers type annotations taking into consideration control-flow and alias information, (2) The interprocedural phase defines unification constraints and propagates them across function boundaries.

# Chapter 5

## Static Detection of Runtime Errors

### 5.1 Introduction

The objective of the current chapter is to ensure that our static safety analysis defined in the previous chapter is able to catch runtime errors caused by its targeted set of unsafe memory operations. To this end, we define an operational semantics of our imperative language defined in Section 4.2.1 that complies with the ANSI C standard. Besides, the semantics evaluates standard undefined behaviours to runtime errors. They encompass accessing uninitialized pointers, dereferencing null pointers, freeing unallocated pointers, etc. Notice that we focus on undefined behaviours that can statically be detected by our memory safety checks defined in Chapter 4.

In this chapter, we show that the dynamic semantics computes expression values that are consistent with the types assigned to them statically. We establish the soundness of our type and effect analysis in detecting memory errors. Since we strive

for soundness, our analysis tends to generate false positives that affect its precision. As a mean to enhance the latter, our analysis provides an effect-based interface for exporting *dunno points* that pinpoint the location and the program traces of suspicious operations that should be considered using dynamic analysis.

The chapter is organized as follows: Section 5.2 presents our ANSI C compliant operational semantics. The static semantics and dynamic semantics are proved consistent in Section 5.3. Based on the consistency results, Section 5.4 shows the soundness of our static analysis for memory error detection. Section 5.4 is dedicated to the effect-based interface for guiding dynamic analysis. To demonstrate the feasibility and the efficiency of our safety evaluation approach, we present our implemented prototype and experimental results in Section 5.6. We conclude this chapter in Section 5.7.

## 5.2 Dynamic Semantics

The dynamic semantics of our imperative language is specified by the means of a big-step structural operational semantics [67, 102, 103] that complies with the ANSI C standard [75]. We list in Table 11 the computable values that are derived from the evaluation of program expressions by our dynamic semantics. We consider the command value `unit`, the undefined value `undef` that represents the value of uninitialized variables, the integer value `int`, memory locations `loc`, and memory blocks for a structure `loc@⟨ $v_i$ ⟩1..n` where each  $v_i$  corresponds to a member of the structure. According to the standard, members of a structure are stored in the order they are

declared in their corresponding structure type. We also consider function closures  $\langle id, x, s_{id}, E \rangle$  in order to formally capture call-time environments. A closure is composed of the function identifier  $id$ , the function argument  $x$ , the function statements  $s_{id}$ , and an environment  $E$  where it is defined. In fact, an environment  $E$  maps variable identifiers to memory locations containing the values of these variables. We also define a store  $\mathcal{C}$  that maps memory locations of variables and dynamically allocated locations to values. A trace  $f$  represents the side-effects of memory management operations.

---

**Table 11** Computable values

---

$loc$	$\in$	$Ref$	
$v$	$\in$	$Value$	= $unit$
			$undef$
			$loc$
			$int$
			$loc@ \langle v_1, \dots, v_n \rangle$
			$\langle id, x, s_{id}, E \rangle$
$E$	$\in$	$Env$	= $Id \rightarrow Ref$
$\mathcal{C}$	$\in$	$Store$	= $Ref \rightarrow Value$
$f$	$\in$	$Trace$	= $\emptyset$
			$f; f'$
			$alloc(loc)$
			$dealloc(loc)$
			$read(loc)$
			$assign(loc)$
			$arith(loc)$
			$stack(loc)$

---

The operational rules presented in Table 12 and Table 13 specify how to execute a program in our source language. The rules are defined by the following judgments:

- $\mathcal{C}, E \vdash e \rightarrow v, f, \mathcal{C}'$  for expressions in r-value position.

- $\mathcal{C}, E \vdash_{lexpr} e \rightarrow v, f, \mathcal{C}'$  for expressions in l-value position.
- $\mathcal{C}, E \vdash s \rightarrow v, f, \mathcal{C}'$  for statement evaluation.

Given a store  $\mathcal{C}$  and an environment  $E$ , each judgment associates a syntactic element to the result  $v$  of its execution, the trace  $f$  of the side-effects generated during the execution, as well as the updated memory state  $\mathcal{C}'$ . The execution of a statement implicitly transfers the control to the next program point. Notice that we distinguish between the execution of an expression in l-value position and an expression in r-value position. The former returns the memory location `loc` that holds the value of the expression, whereas the latter results in the value  $v$  of the expression. Through this chapter, we will write  $m \dagger m'$  to denote the overwriting of any map  $m$  by  $m'$ , i.e., the domain of  $m \dagger m'$  is  $\text{Dom}(m) \cup \text{Dom}(m')$ , and we have  $(m \dagger m')(x) = m'(x)$  if  $x \in \text{Dom}(m')$  and  $m(x)$  otherwise. We will write  $m_{x_1, x_2, \dots}$  to denote the map  $m$  excluding the associations of the form  $x_i \mapsto \_$ .

Our rules are inspired by existing operational semantics of the C programming language [9]. We discuss some of the program expression rules of Table 12 in the following paragraphs. The (Ref) rule evaluates the operand  $l_v$  of `&` to its location `loc`, and derives a pointer to that location. The (Cast) rule is trivial and does not modify the value of the converted expression. The type destination  $\underline{\kappa}$  should be an integer type `int` or a pointer type `ref(⋯)` as entailed by the standard. The (Arith) rule evaluates pointer arithmetic in a pessimistic way stating that the result of the evaluation is undefined. In the absence of dynamic bounds checking, there is

---

**Table 12** Operational semantics for expressions
 

---

<b>Expressions</b>	
$\frac{x \in \text{Dom}(E)}{\mathcal{C}, E \vdash x \rightarrow \mathcal{C}(E(x)), \emptyset, \mathcal{C}}$	(Var)
$\frac{}{\mathcal{C}, E \vdash n \rightarrow \text{int}, \emptyset, \mathcal{C}}$	(Int)
$\frac{\mathcal{C}, E \vdash_{\text{lexp}} l_v \rightarrow \text{loc}, f, \mathcal{C}'}{\mathcal{C}, E \vdash \&l_v \rightarrow \text{loc}, (f; \text{stack}(\text{loc})), \mathcal{C}'}$	(Ref)
$\frac{\mathcal{C}, E \vdash e \rightarrow \text{loc}, f, \mathcal{C}'}{\mathcal{C}, E \vdash *e \rightarrow \mathcal{C}'(\text{loc}), (f; \text{read}(\text{loc})), \mathcal{C}'}$	(Deref)
$\frac{\mathcal{C}, E \vdash e \rightarrow v, f, \mathcal{C}' \quad \kappa \subseteq \{\text{int}, \text{ref}(\_)\}}{\mathcal{C}, E \vdash (\kappa)e \rightarrow v, f, \mathcal{C}'}$	(cast)
$\frac{\mathcal{C}, E \vdash e \rightarrow \text{loc}@(\varphi_i)_{i=1..n}, f, \mathcal{C}' \quad \mathcal{C}'(\text{loc} + \text{offset}(\varphi_i)) = v_i}{\mathcal{C}, E \vdash e.\varphi_i \rightarrow v_i, f, \mathcal{C}'}$	(Field)
$\frac{\mathcal{C}, E \vdash e' \rightarrow \text{loc}, f', \mathcal{C}' \quad \mathcal{C}', E \vdash e'' \rightarrow \text{int}, f'', \mathcal{C}''}{\mathcal{C}, E \vdash e' \oplus e'' \rightarrow \text{undef}, (f'; f''; \text{arith}(\text{loc})), \mathcal{C}''}$	(Arith)
$\frac{\mathcal{C}, E \vdash e \rightarrow \text{int}, f, \mathcal{C}' \quad \text{loc fresh} \quad f' = f; \text{alloc}(\text{loc})}{\mathcal{C}, E \vdash \text{malloc}(e) \rightarrow \text{loc}, f', \mathcal{C}' \uparrow [\text{loc} \mapsto \text{undef}]}$	(Malloc)

---

no guarantee that the resulting pointer will remain within its assigned boundaries. Moreover, the C standard [75] states that the dereference of out-of-bounds pointers generates undefined behaviours that we consider as runtime errors. By conservatively setting the value of pointer arithmetic to undef, we are able to capture any potential and undesirable out-of-bounds access. The (Malloc) rule returns a fresh location `loc` that contains an undefined value as stated by the C standard [75].

The operational rules for program statements are listed in Table 13. The (Free) rule deallocates the memory location of a pointer that is previously returned by a dynamic allocation operation. The content of a freed location `loc` is undefined since

---

**Table 13** Operational semantics for statements
 

---

Statements	
$\frac{C, E \vdash e \rightarrow \text{loc}, f, C' \quad (\text{stack}(\text{loc}) \notin f) \quad \text{Dom}(C') = \text{Dom}(C'')}{C, E \vdash \text{free}(e) \rightarrow \text{unit}, (f; \text{dealloc}(\text{loc})), C''_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}]}$ $C''(\text{loc}') = \begin{cases} \text{undef} & \text{if } C'(\text{loc}') = \text{loc}, \\ C'(\text{loc}') & \text{otherwise.} \end{cases}$	(Free)
$\frac{C_0, E \vdash_{\text{lexpr}} e \rightarrow \text{loc}, f, C \quad C, E \vdash e' \rightarrow v, f', C' \quad ((v = \text{int}) \vee (v \subseteq \text{Ref})) \quad f'' = f; f'; \text{assign}(\text{loc})}{C_0, E \vdash e = e' \rightarrow v, f'', C''_{\text{loc}} \dagger [\text{loc} \mapsto v]}$	(Assign)
$\frac{C_0, E \vdash_{\text{lexpr}} e \rightarrow \text{loc}, f, C_1 \quad C_1, E \vdash \text{call}_{id} \rightarrow (id, x, s_{id}, E'), \emptyset, C_2 \quad C_2, E \vdash e' \rightarrow v', f', C_3 \quad C_3 \dagger [E'(x) \mapsto v'], E' \vdash s_{id} \rightarrow v'', f'', C'}{C_0, E \vdash e = id(e') \rightarrow v'', (f; f'; f''), C''_{\text{loc}} \dagger [\text{loc} \mapsto v'']}$	(call)
$\frac{C, E \vdash e \rightarrow v, f, C'}{C, E \vdash \text{return } e \rightarrow v, f, C'}$	(Return)
$\frac{C_0, E \vdash s \rightarrow v, f, C \quad C, E \vdash s' \rightarrow v', f', C'}{C_0, E \vdash s; s' \rightarrow v', (f; f'), C'}$	(Seq)
$\frac{C_0, E \vdash e \rightarrow v, f, C \quad v \neq 0 \quad C, E \vdash s' \rightarrow v', f', C'}{C_0, E \vdash \text{if } e \text{ then } s' \text{ else } s'' \rightarrow v', (f; f'), C'}$	(If-T)
$\frac{C_0, E \vdash e \rightarrow v, f, C \quad v = 0 \quad C, E \vdash s'' \rightarrow v'', f'', C''}{C_0, E \vdash \text{if } e \text{ then } s' \text{ else } s'' \rightarrow v'', (f; f''), C''}$	(If-F)
$\frac{C_0, E \vdash e \rightarrow v, f, C \quad v \neq 0 \quad C, E \vdash s; \text{while } (e) \text{ do } s \rightarrow v', f', C'}{C_0, E \vdash \text{while } (e) \text{ do } s \rightarrow v', (f; f'), C'}$	(While-T)
$\frac{C, E \vdash e \rightarrow v, f, C' \quad v = 0}{C, E \vdash \text{while } (e) \text{ do } s \rightarrow \text{unit}, f, C'}$	(While-F)

---

loc may be assigned to another process. Besides, all locations that refer to the freed location loc are mapped to an undefined value in the derived store  $\mathcal{C}''$ . The (Assign) rule evaluates the left-hand-side operand to its location loc and updates the store  $\mathcal{C}$  with a mapping from loc to the value of the right-hand-side operand. The (call) rule evaluates the label  $call_{id}$  corresponding to the call to function  $id$ , and yields the function closure  $\langle id, x, s_{id}, E' \rangle$ . The location  $E'(x)$  of the formal argument  $x$  is assigned the value  $v'$  of the actual argument  $e$  in order to evaluate the function statements  $s_{id}$ .

### 5.3 Consistency of Static and Dynamic Semantics

In this section, we prove the consistency of our static and dynamic semantics. We use the proof method that is introduced by Talpin and Jouvelot in [121] to show that the static and the dynamic semantics are consistent with respect to a structural relation between values and types defined as the maximal fixed point of a monotonic property. The result of the consistency is then used to show that our defined static checks report occurrences of all targeted errors and do not suffer false negatives.

We define below a store model  $\mathcal{S}$  that relates a region  $\rho$  and a type  $\tau$  to their corresponding reference value loc.

**Definition 1 (Store Model)** *A store model  $\mathcal{S}$  is a finite mapping from locations loc to pairs  $(\rho, \tau)$  of regions  $\rho$  and types  $\tau$  as follows:*

$$\mathcal{S} \in \text{StoreModel} = \text{Ref} \rightarrow \text{Region} \times \text{Type}$$

*We say that  $\mathcal{S}'$  extends  $\mathcal{S}$ , denoted by  $\mathcal{S} \subseteq \mathcal{S}'$ , if and only if  $\forall \text{loc} \in \text{Dom}(\mathcal{S})$ , we have  $\mathcal{S}(\text{loc}) = \mathcal{S}'(\text{loc})$ .*



In what follows, we define the relations  $\mathcal{S} \models f : \sigma$  and  $\mathcal{C} : \mathcal{S} \models v : \tau$  that we need to establish a link between the static and the dynamic semantics.

**Definition 2 (Effect Consistency)** *A dynamic trace of side-effects  $f \in \text{Trace}$  is consistent with the effect  $\sigma \in \text{Effect}$  for the model  $\mathcal{S} \in \text{StoreModel}$ , noted  $\mathcal{S} \models f : \sigma$ , if and only if:*

$$\begin{aligned} \forall \text{alloc}(\text{loc}) \in f, \mathcal{S}(\text{loc}) = (\rho, \tau) \wedge \text{alloc}(\rho, \ell) \in \sigma \\ \forall \text{free}(\text{loc}) \in f, \mathcal{S}(\text{loc}) = (\rho, \tau) \wedge \text{free}(\rho, \ell) \in \sigma \\ \forall \text{read}(\text{loc}) \in f, \mathcal{S}(\text{loc}) = (\rho, \tau) \wedge \text{read}(\rho, \tau, \ell) \in \sigma \\ \forall \text{assign}(\text{loc}) \in f, \mathcal{S}(\text{loc}) = (\rho, \tau) \wedge \text{assign}(\rho, \tau, \ell) \in \sigma \\ \forall \text{arith}(\text{loc}) \in f, \mathcal{S}(\text{loc}) = (\rho, \tau) \wedge \text{arith}(\rho, \ell) \in \sigma \\ \forall \text{stack}(\text{loc}) \in f, \mathcal{S}(\text{loc}) = (\rho, \tau) \wedge \text{stack}(\rho, \ell) \in \sigma \end{aligned}$$

The definition relates a dynamic effect on a location  $\text{loc}$  to a static effect on a region  $\rho$  and a type  $\tau$  that correspond to  $\text{loc}$  through the store model  $\mathcal{S}$ . Note that we also have the following:

- If  $\mathcal{S} \subseteq \mathcal{S}'$  and  $\mathcal{S} \models f : \sigma$ , then  $\mathcal{S}' \models f : \sigma$ .
- If  $\mathcal{S} \models f : \sigma$  and  $\mathcal{S} \models f' : \sigma'$  then  $\mathcal{S} \models f; f' : \sigma; \sigma'$ .
- If  $\mathcal{S} \models f : \sigma$  then  $\forall \sigma'$ , we have  $\mathcal{S} \models f : \text{if}(\sigma, \sigma')$  and  $\mathcal{S} \models f : \text{if}(\sigma', \sigma)$ .

We define typed stores as models for describing the relation between values and types.

$$\mathcal{C} : \mathcal{S} \in \text{TypedStore} = \text{Store} \times \text{StoreModel}$$

Given a typed store  $\mathcal{C} : \mathcal{S}$ , the following definition establishes the link between values of expressions computed by the dynamic semantics to their types evaluated by the static counterpart.

**Definition 3 (Consistent Values and Types)** *Given a typed store  $\mathcal{C} : \mathcal{S}$ , the value  $v$  is consistent with the type  $\tau$ , noted  $\mathcal{C} : \mathcal{S} \models v : \tau$ , if and only if  $v$  and  $\tau$  verify one of the following properties:*

$$\begin{aligned}
\mathcal{C} : \mathcal{S} \models \text{unit} & : \text{unit} \\
\mathcal{C} : \mathcal{S} \models \text{int} & : \text{int}_{[\&\text{int}]} \\
\mathcal{C} : \mathcal{S} \models \text{undef} : \tau & \Leftrightarrow \bar{\tau} \subseteq \{\text{void}, \text{ref}(\text{void})\} \text{ or } \text{hostOf}(\tau) \cap \{\text{[wild]}, \text{[dangling]}, \text{[arith]}\} \neq \emptyset \\
\mathcal{C} : \mathcal{S} \models \text{loc} : \tau & \Leftrightarrow \mathcal{S}(\text{loc}) = (\text{regionOf}(\tau), \text{strTypeOf}(\tau)) \\
& \text{ and } \mathcal{C} : \mathcal{S} \models \mathcal{C}(\text{loc}) : \text{strTypeOf}(\tau) \\
\mathcal{C} : \mathcal{S} \models \text{loc} : \text{int}_{[\&\tau]} & \Leftrightarrow \bar{\tau} = \text{ref}(\_) \text{ and } \mathcal{C} : \mathcal{S} \models \text{loc} : \tau \\
\mathcal{C} : \mathcal{S} \models \text{loc}@ \langle v_i \rangle_{1..n} : \tau & \Leftrightarrow \forall \varphi_i \in \text{fldList}(\tau) \text{ and } \text{loc}_i = \text{loc} + \text{offset}(\varphi_i), \\
& \mathcal{C}(\text{loc}_i) = v_i \text{ and } \mathcal{C} : \mathcal{S} \models \mathcal{C}(\text{loc}_i) : \text{fldType}(\tau, \varphi_i) \\
\mathcal{C} : \mathcal{S} \models \langle \text{id}, x, s_{\text{id}}, E \rangle : \tau & \Leftrightarrow \exists \mathcal{E} \text{ such that } \mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{E}, \ell \vdash \text{call}_{\text{id}} : \tau, \mathcal{E}'
\end{aligned}$$

The intuition behind this definition is that the static type of a given expression is a conservative estimation of its corresponding value derived from the dynamic semantics. In the dynamic semantics, the `undef` value stands for a program expression that has an uninitialized value. In the static counterpart, that same expression can either be assigned a void type, a void pointer type, a `[wild]` annotated type, a pointer type with `[dangling]` annotation, or a pointer type with `[arith]` annotation. A program expression that is initialized to an integer value `int` should have an integer type `int[\&int]` where the host annotation `[\&int]` indicates an initialized value. A `loc` value corresponds to a pointer type  $\tau$  such that its referred region `regionOf( $\tau$ )` and type `strTypeOf( $\tau$ )` are related to `loc` in the store model  $\mathcal{S}$ . Moreover, the value  $\mathcal{C}(\text{loc})$  stored in `loc` corresponds to the type `strTypeOf( $\tau$ )` referred to by pointer  $\tau$ . In our static semantics, we allow casting from pointer type to integer type provided that the latter is large enough to hold a pointer value. The host annotation of the derived integer type keeps track of its original pointer type. An integer type `int[\&\tau]` where  $\bar{\tau} = \text{ref}(\_)$  is actually a pointer type  $\tau$  in integer disguise. Therefore, a value `loc` is consistent with `int[\&\tau]`, if and only if `loc` is consistent with its hidden pointer type  $\tau$ .

---

**Table 14** Function  $\mathcal{F}$  whose maximal fixed point defines the consistency relation

---


$$\mathcal{F}(\mathcal{Q}) = \{(C : S, v, \tau)\}$$

if  $v = \text{unit}$  then  $\tau = \text{void}$   
 if  $v = \text{int}$  then  $\tau = \text{int}_{\{\&\text{int}\}}$   
 if  $v = \text{undef}$  then  $\text{hostOf}(\tau) \cap \{\{\text{wild}\}, \{\text{dangling}\}, \{\text{arith}\}\} \neq \emptyset$  or  $\tau \in \{\text{void}, \text{ref}(\text{void})\}$   
 if  $v = \text{loc}$  then  $\exists \rho, \tau'$  such that  $\rho = \text{regionOf}(\tau)$  and  $\tau' = \text{strTypeOf}(\tau)$   
     and  $(C : S, \mathcal{C}(v), \tau') \in \mathcal{Q}$  and  $S(\text{loc}) = (\rho, \tau')$   
 if  $v = \text{loc}@ \langle v_i \rangle_{1..n}$  then  $\tau = \text{struct}\{(\varphi_i, \tau_i, o_i)\}_{1..n}$  and  $\forall \text{loc}_i = \text{loc} + \text{offset}(\varphi_i)$ ,  
      $\mathcal{C}(\text{loc}_i) = v_i$  and  $\bigcup_{i=1..n} \{(C : S, v_i, \tau_i)\} \in \mathcal{Q}$   
 if  $v = \langle \text{id}, x, s_{\text{id}}, E \rangle$  then  $\exists \mathcal{E}$  such that  $C : S \models E : \mathcal{E}$  and  $\mathcal{E}, \ell \vdash \text{call}_{\text{id}} : \tau, \mathcal{E}'$

---

We note  $C : S \models E : \mathcal{E}$ , if and only if  $\text{Dom}(E) = \text{Dom}(\mathcal{E})$  and for every  $x \in \text{Dom}(E)$ , we have  $C : S \models \mathcal{C}(E(x)) : \mathcal{E}(x)$ . If  $C : S \models E : \mathcal{E}$  then for every  $x \in \text{Dom}(E)$ , we have  $S(E(x)) = (\text{addressOf}(x), \mathcal{E}(x))$ . We have also the following:

- if  $C : S \models v : \tau$  then  $C : S \models v : \text{if}(\tau, \tau')$  and  $C : S \models v : \text{if}(\tau', \tau)$ .
- If  $C : S \models E : \mathcal{E}$  then  $C : S \models E : \mathcal{E} \mathbb{M} \mathcal{E}'$  and  $C : S \models E : \mathcal{E}' \mathbb{M} \mathcal{E}$

As discussed in [121, 124], this structural property between values and types does not uniquely define a consistency relation and must be regarded as a fixed point equation on the domain  $\mathcal{R} = \text{TypedStore} \times \text{Value} \times \text{Type}$  of the relation. In Table 5.3, we define function  $\mathcal{F}$ , on the domain  $\mathcal{P}_{\text{fn}}(\mathcal{R}) \rightarrow \mathcal{P}_{\text{fn}}(\mathcal{R})$ , whose fixed points are the relations on  $\mathcal{R}$  that verify the property defined above. To ensure the existence of the greatest fixed point  $\text{gfp}(\mathcal{F})$ , function  $\mathcal{F}$  must be monotonic.

**Lemma 5.3.1 (Monotonicity of  $\mathcal{F}$ )** *If  $\mathcal{Q} \subseteq \mathcal{Q}'$  then  $\mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$ .*

**Proof of Monotonicity Lemma 5.3.1** Let  $\mathcal{Q}$  and  $\mathcal{Q}'$  be two subsets of  $\mathcal{R}$  such that  $\mathcal{Q} \subseteq \mathcal{Q}'$ . We assume that  $q \in \mathcal{F}(\mathcal{Q})$  and prove that  $q \in \mathcal{F}(\mathcal{Q}')$ . Let  $q$  be  $(C : S, v, \tau)$ :

- (i) if  $v \in \{\text{unit}, \text{int}, \text{undef}\}$  then  $q \in \mathcal{F}(\mathcal{Q}')$  by definition.
- (ii) if  $v = \text{loc}$ , there exist  $\rho$  and  $\tau'$  such that  $((\rho = \text{regionOf}(\tau)$  and  $\tau' = \text{strTypeOf}(\tau)$ ) or  $(\tau = \text{int}_{\{\&\text{ref}_\rho(\_)_{\{\&\tau'\}}\}}))$  and  $S(v) = (\rho, \tau')$  and  $(C : S, \mathcal{C}(v), \tau') \in \mathcal{Q}$ . Since  $\mathcal{Q} \subseteq \mathcal{Q}'$ , we have  $q \in \mathcal{F}(\mathcal{Q}')$ .

- (iii) if  $v = \text{loc}@ \langle v_i \rangle_{1..n}$ , then  $\tau = \text{struct}\{(\varphi_i, \tau_i, o_i)\}_{1..n}$  and for  $\text{loc}_i = \text{loc} + \text{offset}(\varphi_i)$ , we have  $\mathcal{C}(\text{loc}_i) = v_i$  and  $\bigcup_{i=1..n} (\mathcal{C} : \mathcal{S}, \mathcal{C}(\text{loc}_i), \tau_i) \in \mathcal{Q}$ . Since  $\mathcal{Q} \subseteq \mathcal{Q}'$ , we have  $q \in \mathcal{F}(\mathcal{Q}')$
- (iv) Finally, if  $v = \langle \text{id}, x, s_{\text{id}}, E \rangle$  then there exists a type environment  $\mathcal{E}$  such that  $\mathcal{C} : \mathcal{S} \models E : \mathcal{E}$ , so that  $q \in \mathcal{F}(\mathcal{Q}')$ . ■

Among the fixed points of  $\mathcal{F}$ , we choose the greatest fixed point  $\text{gfp}(\mathcal{F})$  as our consistency relation [121, 124];  $\text{gfp}(\mathcal{F})$  is defined by:

$$\text{gfp}(\mathcal{F}) = \bigcup \{ \mathcal{Q} \subseteq \mathcal{R} \mid \mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q}) \}$$

A set  $\mathcal{Q}$  such that  $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$  is called  $\mathcal{F}$ -consistent. The relation between values and types is thus defined by:

$$\mathcal{C} : \mathcal{S} \models v : \tau \Leftrightarrow (\mathcal{C} : \mathcal{S}, v, \tau) \in \text{gfp}(\mathcal{F})$$

In order to use induction in the consistency proof, we need to check that the relation between a type and a value, whenever correct for some typed store  $\mathcal{C} : \mathcal{S}$ , is preserved when the store is properly expanded. We note:

$$\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}' \Leftrightarrow \mathcal{S} \subseteq \mathcal{S}' \text{ and } \mathcal{C} \subseteq \mathcal{C}', \text{ and for all } v \text{ and } \tau, \mathcal{C} : \mathcal{S} \models v : \tau \Rightarrow \mathcal{C}' : \mathcal{S}' \models v : \tau$$

**Lemma 5.3.2 (Side Effects)** *Assume  $\mathcal{C} : \mathcal{S} \models v : \tau$ . If  $\mathcal{S}(\text{loc}) = (\rho, \tau)$ , then  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}_{\text{loc}} \dagger \{ \text{loc} \mapsto v \} : \mathcal{S}_{\text{loc}} \dagger \{ \text{loc} \mapsto (\rho, \tau) \}$ . Otherwise, for every region  $\rho$ ,  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C} \dagger \{ \text{loc} \mapsto v \} : \mathcal{S} \dagger \{ \text{loc} \mapsto (\rho, \tau) \}$ .*

**Proof of Side Effects Lemma 5.3.2** The proof is done by induction on the structures of typing and values. Let  $\mathcal{C}' = \mathcal{C}_{\text{loc}} \dagger \{ \text{loc} \mapsto v \}$  and  $\mathcal{S}'_{\text{loc}} = \mathcal{S}_{\text{loc}} \dagger \{ \text{loc} \mapsto (\rho, \tau) \}$ . We have to prove that  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}'$ , i.e.,  $\mathcal{C}' : \mathcal{S}' \models v : \tau$  given that  $\mathcal{C} : \mathcal{S} \models v : \tau$ ,  $\mathcal{C} \subseteq \mathcal{C}'$ , and  $\mathcal{S} \subseteq \mathcal{S}'$ . Considering the typed store  $\mathcal{C} : \mathcal{S}$  and  $\mathcal{Q} \subseteq \mathcal{R}$  such that  $\mathcal{Q} = \{ (\mathcal{C}' : \mathcal{S}', v, \tau) \mid \mathcal{C} : \mathcal{S} \models v : \tau \}$ , we show that  $\mathcal{Q}$  is  $\mathcal{F}$ -consistent, i.e.,  $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$ . Let  $q = (\mathcal{C}' : \mathcal{S}', v, \tau) \in \mathcal{Q}$ :

- (i) if  $v \in \{ \text{unit}, \text{int}, \text{undef} \}$  then  $q \in \mathcal{F}(\mathcal{Q})$

- (ii) if  $v = \text{loc}'$ , by the definition of  $\mathcal{C} : \mathcal{S} \models v : \tau$ , there exist  $\rho'$  and  $\tau'$  such that  $((\rho' = \text{regionOf}(\tau) \text{ and } \tau' = \text{strTypeOf}(\tau)) \text{ or } (\tau = \text{int}_{[\&\text{ref}_{\rho'}(\_)_{\&\tau'}]})$ ) and  $\mathcal{S}(v) = (\rho', \tau')$  and  $\mathcal{C} : \mathcal{S} \models \mathcal{C}(v) : \tau'$ . Since  $\mathcal{C} \subseteq \mathcal{C}'$  and  $\mathcal{S} \subseteq \mathcal{S}'$  then  $\mathcal{S}'(v) = (\rho', \tau')$  and  $\mathcal{C} : \mathcal{S} \models \mathcal{C}'(v) : \tau'$ , so that  $(\mathcal{C}' : \mathcal{S}', \mathcal{C}'(v), \tau') \in \mathcal{Q}$  and  $q \in \mathcal{F}(\mathcal{Q})$
- (iii) if  $v = \text{loc}@ \langle v_i \rangle_{1..n}$ , by definition of  $\mathcal{C} : \mathcal{S} \models v : \tau$ , we have  $\tau = \text{struct}\{(\varphi_i, \tau_i, o_i)\}$ . Let  $\text{loc}_i = \text{loc} + \text{offset}(\varphi_i)$ , we have  $\mathcal{C}(\text{loc}_i) = v_i$  and  $\mathcal{C} : \mathcal{S} \models v_i : \tau_i$ . Since  $\mathcal{C} \subseteq \mathcal{C}'$  and  $\mathcal{S} \subseteq \mathcal{S}'$  then  $\mathcal{C} : \mathcal{S} \models \mathcal{C}'(\text{loc}_i) : \tau_i$ , and  $\bigcup_{i=1..n} (\mathcal{C}' : \mathcal{S}', \mathcal{C}'(\text{loc}_i), \tau_i) \in \mathcal{Q}$  and  $q \in \mathcal{F}(\mathcal{Q})$
- (iv) Finally, if  $v = \langle \text{id}, x, s_{\text{id}}, E \rangle$  then there exists a type environment  $\mathcal{E}$  such that  $\mathcal{C} : \mathcal{S} \models E : \mathcal{E}$ . This means that  $\mathcal{C} : \mathcal{S} \models \mathcal{C}(E(x)) : \mathcal{E}(x)$  for every  $x \in \text{Dom}(E)$ . Thus, we have  $(\mathcal{C}' : \mathcal{S}', \mathcal{C}(E(x)), \mathcal{E}(x)) \in \mathcal{Q}$ , so that  $q \in \mathcal{F}(\mathcal{Q})$ .  $\blacksquare$

Lemma 5.3.2 covers the cases of creation of new references and of assigning a value to an existing reference. In both cases, we make sure that the store expansion preserves the consistency relation. Note that if  $\mathcal{C} : \mathcal{S} \models v : \tau$  and  $\mathcal{C} : \mathcal{S} \models E : \mathcal{E}$  then  $\mathcal{C} \dagger [\text{loc} \mapsto \tau] : \mathcal{S} \dagger [\rho \mapsto (\text{loc}, \tau)] \models E : \text{directUpd}(\mathcal{E}, l_v, \tau)$  where  $\text{addressOf}(l_v, \mathcal{E}) = \rho$ . As defined in Chapter 4, function  $\text{directUpd}(\mathcal{E}, l_v, \tau)$  derives an updated environment  $\mathcal{E}'$  where the host annotations of all variables that directly or indirectly refer to  $\rho$  are set to  $[\&\tau]$ .

We also need to check that the relation between a type  $\tau$  and a value  $v$ , whenever correct for some typed store  $\mathcal{C} : \mathcal{S}$ , is preserved when the type is cast to another type  $\tau'$  such that  $\tau' = \text{castType}(\tau, \kappa)$  for all declared type  $\kappa$ . In the dynamic semantics, we only consider type conversions applied on scalar types (integers and pointers).

**Lemma 5.3.3 (Type Conversion)** *Given a typed store  $\mathcal{C} : \mathcal{S}$  and a value  $v$  that is consistent with type  $\tau$ , then  $\mathcal{C} : \mathcal{S} \models v : \tau \Rightarrow \mathcal{C} : \mathcal{S} \models v : \text{castType}(\tau, \kappa)$  for all  $\kappa$ .*

**Proof of Type Conversion Lemma 5.3.3** The proof of this lemma is done by covering all cases of function  $\text{castType}(\tau, \kappa)$ .

- Let  $v = \text{loc}$ ,  $\tau = \text{ref}_{\rho}(\kappa')_{\eta}$  and  $\kappa = \text{int}$ . We have  $\text{castType}(\tau, \text{int}) = \text{int}_{[\&\tau]}$ . Since  $\mathcal{C} : \mathcal{S} \models v : \tau$  and  $\bar{\tau} = \text{ref}$ , we have from the  $\models$  relation  $\mathcal{C} : \mathcal{S} \models v : \text{castType}(\tau, \text{int})$ .

- Let  $v = \text{loc}$ ,  $\tau = \text{ref}_\rho(\text{void})_{[\text{malloc}]}$  and  $\kappa = \text{ref}(\kappa')$ . We have  $\text{castType}(\tau, \text{ref}(\kappa')) = \tau'$  with  $\tau' = \text{ref}_\rho(\kappa')_{[\&\kappa']}$  and  $\text{strTypeOf}(\tau) = \text{strTypeOf}(\tau')$  and  $\text{regionOf}(\tau) = \text{regionOf}(\tau')$ . From the  $\models$  relation, we conclude that  $\mathcal{C} : \mathcal{S} \models v : \tau \Rightarrow \mathcal{C} : \mathcal{S} \models v : \text{castType}(\tau, \text{ref}(\kappa'))$ .
- Let  $v = \text{loc}$ ,  $\tau = \text{ref}_\rho(\kappa')_\eta$  and  $\kappa = \text{ref}(\kappa'')$ . We have  $\text{castType}(\tau, \text{ref}(\kappa'')) = \tau'$  with  $\tau' = \text{ref}_\rho(\kappa'')_\eta$  and  $\text{strTypeOf}(\tau) = \text{strTypeOf}(\tau')$  and  $\text{regionOf}(\tau) = \text{regionOf}(\tau')$ . From the  $\models$  relation, we conclude that  $\mathcal{C} : \mathcal{S} \models v : \tau \Rightarrow \mathcal{C} : \mathcal{S} \models v : \text{castType}(\tau, \text{ref}(\kappa''))$ .
- Let  $v = \text{loc}$ ,  $\tau = \text{int}_{[\&\text{ref}_\rho(\kappa')_\eta]}$  and  $\kappa = \text{ref}(\kappa'')$ . We have  $\text{castType}(\tau, \text{ref}(\kappa'')) = \text{ref}_\rho(\kappa'')_\eta$ . Since  $\mathcal{C} : \mathcal{S} \models v : \tau$ , we have  $\mathcal{C} : \mathcal{S} \models v : \text{ref}_\rho(\kappa'')_\eta$  and  $\mathcal{C} : \mathcal{S} \models v : \text{castType}(\tau, \text{ref}(\kappa''))$ .
- Let  $v = \text{int}$ ,  $\tau = \text{int}_{[\&\text{int}]}$ . For all  $\kappa$ , we have  $\text{castType}(\tau, \kappa) = \tau$  and we conclude that  $\mathcal{C} : \mathcal{S} \models v : \tau \Rightarrow \mathcal{C} : \mathcal{S} \models v : \text{castType}(\tau, \kappa)$ .
- Let  $v = \text{undef}$ , then  $\tau$  is either of the form  $\text{int}_{[\text{wild}]}$ ,  $\text{ref}_\rho(\_)_{[\text{dangling}]}$ ,  $\text{ref}_\rho(\_)_{[\text{wild}]}$ ,  $\text{ref}_\rho(\_)_{[\text{arith}]}$ ,  $\text{ref}_\rho(\text{void})_{[\text{malloc}]}$ , or  $\text{void}$ . For all  $\kappa$ , we have  $\text{castType}(\tau, \kappa) = \tau$  and we conclude that  $\mathcal{C} : \mathcal{S} \models v : \tau \Rightarrow \mathcal{C} : \mathcal{S} \models v : \text{castType}(\tau, \kappa)$ . ■

Lemma 5.3.3 covers the cases of type conversions. In our dynamic semantics, we assume that a value  $v$  remains the same when converted to another type. Similarly, the region and the host annotations of a pointer type are kept the same when converted to another pointer type. In other words, the region of a pointer type and its actual referred type are unchanged. As such, we can always establish a link between a computable value and a type whether converted or not. The conversion between integer and pointer types are already covered in Definition 3 of the consistency relation.

Now that we have defined the different relations between the static semantics and the dynamic semantics, we can state the consistency Theorem 5.3.4

**Theorem 5.3.4 (Consistency of dynamic and static semantics)** *Let  $E$  be an environment and  $\mathcal{E}$  its type. Let  $\mathcal{C} : \mathcal{S}$  be a typed store such that  $\mathcal{C} : \mathcal{S} \models E : \mathcal{E}$*

(i) Provided that  $\mathcal{E}, \ell \vdash e : \tau, \sigma$  and  $\mathcal{C}, E \vdash e \rightarrow v, f, \mathcal{C}'$ , there exists a store model  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}'$  with:

$$\mathcal{S}' \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}' \models v : \tau$$

(ii) Provided that  $\mathcal{E}, \ell \vdash e : \tau, \sigma$  and  $\mathcal{C}, E \vdash_{\text{lexpr}} e \rightarrow \text{loc}, f, \mathcal{C}'$ , there exists a store model  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}'$  with:

$$\mathcal{S}' \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}' \models \mathcal{C}'(\text{loc}) : \tau \text{ and } \mathcal{S}'(\text{loc}) = (\text{addressOf}(e, \mathcal{E}), \tau)$$

(iii) Provided that  $\mathcal{E}, \ell \vdash s : \tau, \mathcal{E}', \sigma$  and  $\mathcal{C}, E \vdash s \rightarrow v, f, \mathcal{C}'$ , there exists a store model  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}'$  with:

$$\mathcal{S}' \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}' \models v : \tau \text{ and } \mathcal{C}' : \mathcal{S}' \models E : \mathcal{E}'$$

**Proof of Consistency Theorem 5.3.4** The proof is done by structural induction on expressions:

- **Case of (Var):** By hypothesis, we have:

$$\mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{C}, E \vdash x \rightarrow \mathcal{C}(E(x)), \emptyset, \mathcal{C} \text{ and } \mathcal{E}, \ell \vdash x : \tau, \emptyset$$

We must have  $x \in \text{Dom}(\mathcal{E})$  and  $x \in \text{Dom}(E)$ . From  $\mathcal{C} : \mathcal{S} \models E : \mathcal{E}$  and by taking  $\mathcal{S}' = \mathcal{S}$ , we conclude:

$$\mathcal{S} \models \emptyset : \emptyset \text{ and } \mathcal{C} : \mathcal{S} \models \mathcal{C}(E(x)) : \mathcal{E}(x)$$

- **Case of (Int):** By hypothesis, we have:

$$\mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{C}, E \vdash n \rightarrow \text{int}, \emptyset, \mathcal{C} \text{ and } \mathcal{E}, \ell \vdash n : \text{int}_{[\&\text{int}]}, \emptyset$$

By taking  $\mathcal{S}' = \mathcal{S}$ , we conclude:

$$\mathcal{S} \models \emptyset : \emptyset \text{ and } \mathcal{C} : \mathcal{S} \models \text{int} : \text{int}_{[\&\text{int}]}$$

- **Case of (Malloc):** By hypothesis, we have:

$$\mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{C}, E \vdash \text{malloc}(e) \rightarrow \text{loc}, (f; \text{alloc}(\text{loc})), \mathcal{C}' \uparrow [\text{loc} \mapsto \text{undef}] \\ \text{and } \mathcal{E}, \ell \vdash \text{malloc}(e) : \text{ref}_{\rho}(\text{void})_{[\text{malloc}]}, (\sigma; \text{alloc}(\rho, \ell))$$

By definition of the dynamic and the static semantics, this requires that:

$$\mathcal{C}, E \vdash e \rightarrow \text{int}, f, \mathcal{C}' \quad \text{and} \quad \mathcal{E}, \ell \vdash e : \text{int}_{[\&\text{int}]}, \sigma$$

By induction on  $e$ , there exists a store model  $\mathcal{S}_1$  such that  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}_1$  verifying:

$$\mathcal{S}_1 \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}_1 \models \text{int} : \text{int}_{[\&\text{int}]}$$

By Definition 2 of Effect Consistency, we have  $\{\text{loc} \mapsto (\rho, \text{void})\} \models \text{alloc}(\text{loc}) : \text{alloc}(\rho, \ell)$  and  $\mathcal{C}' : \mathcal{S}_1 \models \text{undef} : \text{void}$ . Since  $\text{loc} \notin \text{Dom}(\mathcal{S}')$ , we define  $\mathcal{S}' = \mathcal{S}_1 \dagger [\text{loc} \mapsto (\rho, \text{void})]$ ; we have:

$$\mathcal{C}' : \mathcal{S}_1 \sqsubseteq \mathcal{C}' \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}'$$

By transitivity of  $\sqsubseteq$ , we conclude that:

$$\begin{aligned} \mathcal{S}' \models f; \text{alloc}(\text{loc}) : \sigma; \text{alloc}(\rho, \ell) \text{ and} \\ \mathcal{C}' \dagger \{\text{loc} \mapsto \text{undef}\} : \mathcal{S}' \models \text{loc} : \text{ref}_\rho(\text{void})_{[\text{malloc}]} \end{aligned}$$

- **Case of (Deref):** By hypothesis, we have:

$$\begin{aligned} \mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{C}, E \vdash *e \rightarrow \mathcal{C}'(\text{loc}), (f; \text{read}(\text{loc})), \mathcal{C}' \\ \text{and } \mathcal{E}, \ell \vdash *e : \tau, (\sigma; \text{read}(\rho, \tau, \ell)) \end{aligned}$$

By definition of the static semantics and dynamic semantics, this requires that:

$$\begin{aligned} \mathcal{C}, E \vdash e \rightarrow \text{loc}, f, \mathcal{C}' \text{ and } \mathcal{E}, \ell \vdash e : \tau', \sigma \text{ and } \bar{\tau}' = \text{ref}(\_) \text{ and} \\ \rho = \text{regionOf}(\tau') \text{ and } \tau = \text{strTypeOf}(\tau') \text{ and } \text{drfChk}(\tau') \end{aligned}$$

By induction on  $e$ , there exists a store model  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}'$  verifying:

$$\mathcal{S}' \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}' \models \text{loc} : \tau'$$

By definition,  $\{\text{loc} \mapsto (\rho, \tau)\} \models \text{read}(\text{loc}) : \text{read}(\rho, \tau, \ell)$ . Since  $\mathcal{S}'(\text{loc}) = (\rho, \tau)$ , we conclude:

$$\mathcal{S}' \models f; \text{read}(\text{loc}) : \sigma; \text{read}(\rho, \tau, \ell) \text{ and } \mathcal{C}' : \mathcal{S}' \models \mathcal{C}'(\text{loc}) : \tau$$

- **Case of (Ref):** By hypothesis, we have:

$$\begin{aligned} \mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{C}, E \vdash \&e \rightarrow \text{loc}, (f; \text{stack}(\text{loc})), \mathcal{C}' \text{ and} \\ \mathcal{E}, \ell \vdash \&e : \text{ref}_\rho(\bar{\tau})_{[\&\tau]}, (\sigma; \text{stack}(\rho, \ell)) \end{aligned}$$

By definition of the semantics, this requires that:

$$\mathcal{C}, E \vdash_{\text{lexpr}} e \rightarrow \text{loc}, f, \mathcal{C}' \text{ and } \mathcal{E}, \ell \vdash e : \tau, \sigma \text{ and } \rho = \text{addressOf}(e, \mathcal{E})$$



By induction on  $e$ , there exists a store model  $S'$  such that  $C : S \sqsubseteq C' : S'$  verifying:

$$S' \models f : \sigma \text{ and } C' : S' \models C'(\text{loc}) : \tau \text{ and } S'(\text{loc}) = (\rho, \tau)$$

By taking  $\tau' = \text{ref}_\rho(\bar{\tau})_{[\&\tau]}$  and by definition of  $\models$ , we have:

$$C' : S' \models \text{loc} : \tau' \text{ where } \rho = \text{regionOf}(\tau') \text{ and } \tau = \text{strTypeOf}(\tau')$$

Thus, we conclude:

$$S' \models f; \text{stack}(\text{loc}) : \sigma; \text{stack}(\rho, \ell) \text{ and } C' : S' \models \text{loc} : \text{ref}_\rho(\bar{\tau})_{[\&\tau]}$$

- **Case of (Cast):** By hypothesis, we have:

$$C : S \models E : \mathcal{E} \text{ and } C, E \vdash (\kappa)e \rightarrow v, f, C' \text{ and } \mathcal{E}, \ell \vdash (\kappa)e : \tau', \sigma$$

By definition of the semantics, this requires that:

$$\begin{aligned} &C, E \vdash e \rightarrow v, f, C' \text{ and } \mathcal{E}, \ell \vdash e : \tau, \sigma \\ &\text{and } \tau' = \text{castType}(\tau, \kappa) \text{ and } \text{castchk}(\tau, \kappa) \end{aligned}$$

By induction on  $e$ , there exists a store model  $S'$  such that  $C : S \sqsubseteq C' : S'$  verifying:

$$S' \models f : \sigma \text{ and } C' : S' \models v : \tau$$

By the Type Conversion Lemma 5.3.3, we conclude that:

$$S' \models f : \sigma \text{ and } C' : S' \models v : \tau'$$

- **Case of (Arith):** By hypothesis, we have:

$$\begin{aligned} &C_0 : S \models E : \mathcal{E} \text{ and } C_0, E \vdash e \oplus e' \rightarrow \text{undef}, (f; \text{arith}(\text{loc})), C' \text{ and} \\ &\mathcal{E}, \ell \vdash e \oplus e' : \text{ref}_\rho(\_)_{[\text{arith}]}, (\sigma; \text{arith}(\rho, \ell)) \end{aligned}$$

By definition of the semantics, this requires that:

$$\begin{aligned} &C_0, E \vdash e \rightarrow \text{loc}, f, C \text{ and } C, E \vdash e' \rightarrow v, f', C' \\ &\text{and } \mathcal{E}, \ell \vdash e : \tau, \sigma \text{ and } \mathcal{E}, \ell \vdash e' : \text{int}_\eta, \sigma' \\ &\text{and } \bar{\tau} = \text{ref}(\_) \text{ and } \rho = \text{regionOf}(\tau) \text{ and } \rho' \text{ fresh} \end{aligned}$$

By induction on  $e$  there exists a store model  $S_1$  such that  $C_0 : S \sqsubseteq C : S_1$  verifying:

$$\mathcal{C} : \mathcal{S}_1 \models \text{loc} : \tau \text{ and } \mathcal{S}_1 \models f : \sigma$$

We define  $\tau' = \text{strTypeOf}(\tau)$ , by definition, we have  $\{\text{loc} \mapsto (\rho, \tau')\} \models \text{arith}(\text{loc}) : \text{arith}(\rho, \ell)$ . Since  $[\text{loc} \mapsto (\rho, \tau')] \subseteq \mathcal{S}_1$ , we have:

$$\mathcal{S}_1 \models \text{arith}(\text{loc}) : \text{arith}(\rho, \ell)$$

By induction on  $e'$ , there exists store model  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S}_1 \sqsubseteq \mathcal{C}' : \mathcal{S}'$  verifying:

$$\mathcal{S}' \models f' : \sigma'$$

By Definition 3 of Consistent Values and Types, we have:  $\mathcal{C}' : \mathcal{S}' \models \text{undef} : \text{ref}_{\rho'}(\_)_{[\text{arith}]}$

By transitivity of  $\sqsubseteq$ , we have  $\mathcal{C}_0 : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}'$ , and we conclude:

$$\mathcal{S}' \models f; f'; \text{arith}(\text{loc}) : \sigma; \sigma'; \text{arith}(\rho, \ell) \text{ and } \mathcal{C}' : \mathcal{S}' \models \text{undef} : \text{ref}_{\rho'}(\_)_{[\text{arith}]}$$

- **Case of (Field):** By hypothesis, we have:

$$\begin{aligned} \mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{C}, E \vdash e.\varphi_i \rightarrow v_i, f, \mathcal{C}' \\ \text{and } \mathcal{E}, \ell \vdash e.\varphi_i : \tau_i, \sigma \end{aligned}$$

By definition of the semantics, this requires that:

$$\begin{aligned} \mathcal{C}, E \vdash e \rightarrow \text{loc}@ \langle v_i \rangle_{i=1..n}, f, \mathcal{C}' \text{ and } \mathcal{C}(\text{loc} + \text{offset}(\varphi_i)) = v_i \\ \text{and } \mathcal{E}, \ell \vdash e : \tau, \sigma \text{ and } \bar{\tau} = \text{struct}\{\_ \} \text{ and } \tau_i = \text{fldType}(\tau, \varphi_i) \text{ and } \text{fldChk}(\tau, \varphi_i) \end{aligned}$$

By induction on  $e$  there exists store model  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S} \sqsubseteq \mathcal{C}' : \mathcal{S}'$  verifying:

$$\mathcal{S}' \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}' \models \text{loc}@ \langle v_i \rangle_{i=1..n} : \tau$$

We define  $\text{loc}_i = \text{loc} + \text{offset}(\varphi_i)$ , by definition we have:

$$\mathcal{C}' : \mathcal{S}' \models \mathcal{C}(\text{loc}_i) : \tau_i$$

Thus, we conclude:

$$\mathcal{S}' \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}' \models v_i : \tau_i$$

- **Case of (Free):** By hypothesis, we have:

$$\begin{aligned} \mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and } \mathcal{C}, E \vdash \text{free}(e) \rightarrow \text{unit}, f', \mathcal{C}''_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] \\ \text{and } \mathcal{E}, \ell \vdash \text{free}(e) : \text{void}, \mathcal{E}', (\sigma; \text{dealloc}(\rho, \ell)) \end{aligned}$$

By definition of the operational semantics, this requires that:

$$C, E \vdash e \rightarrow \text{loc}, f, C' \text{ and } f' = f; \text{dealloc}(\text{loc})$$

$$C''(\text{loc}') = \begin{cases} \text{undef} & \text{if } C'(\text{loc}') = \text{loc}, \\ C'(\text{loc}') & \text{otherwise.} \end{cases}$$

By definition of the static semantics, this requires that:

$$\mathcal{E}, \ell \vdash e : \tau, \sigma \text{ and } \bar{\tau} = \text{ref}(\_) \text{ and } \rho = \text{regionOf}(\tau)$$

$$\text{and } \mathcal{E}' = \text{directUpd}(\mathcal{E}, l_v, \text{updHost}(\tau, [\text{dangling}])))$$

By induction on  $e$  there exists store model  $\mathcal{S}_1$  such that  $C : \mathcal{S} \models C' : \mathcal{S}_1$  verifying:

$$\mathcal{S}_1 \models f : \sigma \text{ and } C' : \mathcal{S}_1 \models \text{loc} : \tau$$

We define  $\mathcal{S}''$  with  $\text{Dom}(\mathcal{S}'') = \text{Dom}(\mathcal{S}_1)$  such that:

$$\mathcal{S}''(\text{loc}') = \begin{cases} (\rho', \text{updHost}(\tau, [\text{dangling}])) & \text{if } C'(\text{loc}') = \text{loc} \text{ and } \mathcal{S}_1(\text{loc}') = (\rho', \tau), \\ \mathcal{S}_1(\text{loc}') & \text{otherwise.} \end{cases}$$

By the Side Effects Lemma 5.3.2, we have:  $C' : \mathcal{S}_1 \sqsubseteq C'' : \mathcal{S}''$

Since  $\mathcal{S}''(\text{loc}) = (\rho, \text{strTypeOf}(\tau))$ , by the Side Effects Lemma 5.3.2 we have:  $C'' : \mathcal{S}'' \sqsubseteq C''_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}''_{\text{loc}} \dagger [\text{loc} \mapsto (\rho, \text{void})]$ . By taking  $\mathcal{S}' = \mathcal{S}''_{\text{loc}} \dagger [\text{loc} \mapsto (\rho, \text{void})]$  we have:

$$C : \mathcal{S} \sqsubseteq C''_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}' \text{ and } \mathcal{S}' \models \text{dealloc}(\text{loc}) : \text{dealloc}(\rho, \text{void})$$

By transitivity of  $\sqsubseteq$ , and by the Side Effects Lemma 5.3.2:

$$C''_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}' \models E : \text{directUpd}(\mathcal{E}, l_v, \text{updHost}(\tau, [\text{dangling}])))$$

We conclude:

$$\mathcal{S}' \models f; \text{dealloc}(\text{loc}) : \sigma; \text{dealloc}(\rho, \ell) \text{ and}$$

$$C''_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}' \models \text{unit} : \text{void} \text{ and } C''_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}' \models E : \mathcal{E}'$$

- **Case of (Free):** By hypothesis, we have:

$$C : \mathcal{S} \models E : \mathcal{E} \text{ and } C, E \vdash \text{free}(e) \rightarrow \text{unit}, f', C'_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}]$$

$$\text{and } \mathcal{E}, \ell \vdash \text{free}(e) : \text{void}, \mathcal{E}', (\sigma; \text{dealloc}(\rho, \ell))$$

By definition of the semantics, this requires that:

$$\begin{aligned}
& \mathcal{C}, E \vdash e \rightarrow \text{loc}, f, \mathcal{C}' \text{ and } f' = f; \text{dealloc}(\text{loc}) \\
& \mathcal{E}, \ell \vdash e : \tau, \sigma \text{ and } \bar{\tau} = \text{ref}(\_) \text{ and } \rho = \text{regionOf}(\tau) \\
& \text{and } \mathcal{E}' = \text{directUpd}(\mathcal{E}, l_v, \text{updHost}(\tau, [\text{dangling}]))
\end{aligned}$$

By induction on  $e$  there exists store model  $\mathcal{S}_1$  such that  $\mathcal{C} : \mathcal{S} \models \mathcal{C}' : \mathcal{S}_1$  verifying:

$$\mathcal{S}_1 \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}_1 \models \text{loc} : \tau$$

By definition,  $\{\text{loc} \mapsto (\rho, \text{void})\} \models \text{dealloc}(\text{loc}) : \text{dealloc}(\rho, \ell)$ . Since  $\text{loc} \in \text{Dom}(\mathcal{S}_1)$ , we define  $\mathcal{S}' = \mathcal{S}_1 \dagger [\text{loc} \mapsto (\rho, \text{void})]$

$$\mathcal{C}' : \mathcal{S}_1 \sqsubseteq \mathcal{C}'_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}'$$

By transitivity of  $\sqsubseteq$ , and by the Side Effects Lemma 5.3.2:

$$\mathcal{C}'_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}' \models E : \text{directUpd}(\mathcal{E}, l_v, \text{updHost}(\tau, [\text{dangling}]))$$

We conclude:

$$\begin{aligned}
& \mathcal{S}' \models f; \text{dealloc}(\text{loc}) : \sigma; \text{dealloc}(\rho, \ell) \text{ and} \\
& \mathcal{C}'_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}' \models \text{unit} : \text{void} \text{ and } \mathcal{C}'_{\text{loc}} \dagger [\text{loc} \mapsto \text{undef}] : \mathcal{S}' \models E : \mathcal{E}'
\end{aligned}$$

- **Case of (Assign):** By hypothesis, we have:

$$\begin{aligned}
& \mathcal{C} : \mathcal{S} \models E : \mathcal{E} \text{ and} \\
& \mathcal{C}, E \vdash e = e' \rightarrow v', (f; f'; \text{assign}(\text{loc})), \mathcal{C}''_{\text{loc}} \dagger [\text{loc} \mapsto v'] \\
& \text{and } \mathcal{E}, \ell \vdash e = e' : \tau', \mathcal{E}', (\sigma; \sigma'; \text{assign}(\rho, \tau', \ell))
\end{aligned}$$

By definition of the semantics, this requires that:

$$\begin{aligned}
& \mathcal{C}, E \vdash_{\text{lexpr}} e \rightarrow \text{loc}, f, \mathcal{C}' \text{ and } \mathcal{C}', E \vdash e' \rightarrow v', f', \mathcal{C}'' \\
& \text{and } \mathcal{E}, \ell \vdash e : \tau, \sigma \text{ and } \mathcal{E}, \ell \vdash e' : \tau', \sigma' \\
& \text{and } \rho = \text{addressOf}(e, \mathcal{E}) \text{ and } \mathcal{E}' = \text{directUpd}(\mathcal{E}, l_v, \tau')
\end{aligned}$$

By induction on  $e$  there exists store model  $\mathcal{S}_1$  such that  $\mathcal{C} : \mathcal{S} \models \mathcal{C}' : \mathcal{S}_1$  verifying:

$$- \quad \mathcal{S}_1 \models f : \sigma \text{ and } \mathcal{C}' : \mathcal{S}_1 \models \mathcal{C}'(\text{loc}) : \tau \text{ and } \mathcal{S}_1(\text{loc}) = (\rho, \tau)$$

By induction on  $e'$  there exists store model  $\mathcal{S}'$  such that  $\mathcal{C}' : \mathcal{S}_1 \sqsubseteq \mathcal{C}'' : \mathcal{S}'$  verifying:

$$\mathcal{S}' \models f' : \sigma' \text{ and } \mathcal{C}'' : \mathcal{S}' \models v' : \tau'$$

By the Side Effects Lemma 5.3.2:

$$C'' : S' \sqsubseteq C''_{\text{loc}} \dagger [\text{loc} \mapsto v'] : S'_{\text{loc}} \dagger [\text{loc} \mapsto (\rho, \tau')]$$

We have  $\mathcal{E}' = \text{directUpd}(\mathcal{E}, l_v, \tau')$ , by the Side Effects Lemma 5.3.2, we have:

$$C''_{\text{loc}} \dagger [\text{loc} \mapsto v'] : S'_{\text{loc}} \dagger [\text{loc} \mapsto (\rho, \tau')] \models E : \mathcal{E}'$$

By taking  $S'' = S'_{\text{loc}} \dagger [\text{loc} \mapsto (\rho, \tau')]$ , we conclude that:

$$\begin{aligned} S'' \models f; f'; \text{assign}(\text{loc}) : \sigma; \sigma'; \text{assign}(\rho, \tau', \ell) \\ \text{and } C''_{\text{loc}} \dagger [\text{loc} \mapsto v'] : S'' \models v' : \tau' \text{ and } C''_{\text{loc}} \dagger [\text{loc} \mapsto v'] : S'' \models E : \mathcal{E}' \end{aligned}$$

- **Case of (Call):** By hypothesis, we have:

$$\begin{aligned} C : S \models E : \mathcal{E} \\ \text{and } C, E \vdash e = \text{id}(e') \rightarrow v'', (f_0; \text{assign}(\text{loc})), C'_{\text{loc}} \dagger [\text{loc} \mapsto v''] \\ \text{and } \mathcal{E}, \ell \vdash e = \text{id}(e') : \tau'', \mathcal{E}'', (\sigma_0; \text{assign}(\rho, \tau'', \ell)) \end{aligned}$$

By the definition of the (Call) rule, we have:

$$\begin{aligned} \mathcal{E}, \ell \vdash e : \tau, \sigma \text{ and } \mathcal{E}, \ell \vdash e' : \tau', \sigma' \text{ and } \mathcal{E}, \ell \vdash \text{call}_{\text{id}} : \tau' \xrightarrow{\sigma''} \tau'', \mathcal{E}' \text{ and} \\ \sigma_0 = \sigma; \sigma'; \sigma'' \text{ and } \rho = \text{addressOf}(e, \mathcal{E}) \text{ and } \mathcal{E}'' = \text{directUpd}(\mathcal{E}', e, \tau'') \end{aligned}$$

By definition of the (Call) rule in the dynamic semantics, we have:

$$\begin{aligned} C, E \vdash_{\text{lexpr}} e \rightarrow \text{loc}, f, C_1 \\ \text{and } C_1, E \vdash \text{call}_{\text{id}} \rightarrow \langle \text{id}, x, s_{\text{id}}, E' \rangle, \emptyset, C_2 \\ \text{and } C_2, E \vdash e' \rightarrow v', f', C_3 \\ \text{and } C_3 \dagger [E'(x) \mapsto v'], E' \vdash s_{\text{id}} \rightarrow v'', f'', C' \\ \text{and } f_0 = f; f'; f'' \end{aligned}$$

By induction on  $e$ , there exists a store model  $S_1$  such that  $C : S \sqsubseteq C_1 : S_1$  verifying:

$$S_1 \models f : \sigma \text{ and } C_1 : S_1 \models C_1(\text{loc}) : \tau \text{ and } S_1(\text{loc}) = (\rho, \tau)$$

By induction on  $\text{call}_{\text{id}}$ , there exists a store model  $S_2$  such that  $C_1 : S_1 \sqsubseteq C_2 : S_2$  verifying:

$$C_2 : S_2 \models \langle \text{id}, x, s_{\text{id}}, E' \rangle : \tau' \xrightarrow{\sigma''} \tau'' \text{ and } C_2 : S_2 \models E : \mathcal{E}'$$

By induction on  $e'$ , there exists a store  $S_3$  such that  $C_2 : S_2 \sqsubseteq C_3 : S_3$  verifying:

$$S_3 \models f' : \sigma' \text{ and } C_3 : S_3 \models v' : \tau'$$

By the Side Effects Lemma 5.3.2,  $C_3 : S_3 \models \langle id, x, s_{id}, E' \rangle : \tau' \xrightarrow{\sigma''} \tau''$ . By the definition of  $\models$ , we have  $C_3 : S_3 \models E' : \mathcal{E}$ . We define  $r_x = \text{addressOf}(x, \mathcal{E})$ , by the Side Effects Lemma 5.3.2, we have:

$$C_3 \dagger [E'(x) \mapsto v'] : S_3 \dagger [E'(x) \mapsto (r_x, \tau')] \models E' : \mathcal{E} \dagger [x \mapsto \tau']$$

By induction hypothesis on  $s_{id}$ , there exists a store  $S'$  such that  $C_3 : S_3 \sqsubseteq C' : S'$  which verifies the theorem. Thus,

$$S' \models f'' : \sigma'' \text{ and } C' : S' \models v'' : \tau''$$

From  $C_2 : S_2 \models E : \mathcal{E}'$  and by transitivity of  $\sqsubseteq$ , we have  $C' : S' \models E : \mathcal{E}'$ . By the Side Effects Lemma 5.3.2, we have:

$$C'_{1oc} \dagger [1oc \mapsto v''] : S'_{1oc} \dagger [1oc \mapsto (\rho, \tau'')] \models E : \mathcal{E}'$$

We define  $S'' = S'_{1oc} \dagger [1oc \mapsto (\rho, \tau'')]$ . By the Side Effects Lemma 5.3.2  $C'_{1oc} \dagger [1oc \mapsto v''] : S'' \models E : \text{directUpd}(\mathcal{E}', e, \tau'')$ . Thus, we conclude that:

$$S'' \models f; f'; f''; \text{assign}(1oc) : \sigma; \sigma'; \sigma''; \text{assign}(\rho, \tau'', \ell) \\ \text{and } C'_{1oc} \dagger [1oc \mapsto v''] : S'' \models v'' : \tau'' \text{ and } C'_{1oc} \dagger [1oc \mapsto v''] : S'' \models E : \mathcal{E}''$$

- **Case of (Return):** By hypothesis, we have:

$$C : S \models E : \mathcal{E} \text{ and } C, E \vdash \text{return } e \rightarrow v, f, C \text{ and } \mathcal{E}, \ell \vdash \text{return } e : \tau, \mathcal{E}, \sigma$$

By definition of the semantics, we have:

$$C, E \vdash e \rightarrow v, f, C \text{ and } \mathcal{E}, \ell \vdash e : \tau, \sigma$$

By taking  $S' = S$ , we conclude:

$$S \models f : \sigma \text{ and } C : S \models v : \tau \text{ and } C : S \models E : \mathcal{E}$$

- **Case of (Seq):** By hypothesis, we have:

$$C : S \models E : \mathcal{E} \\ \text{and } C_0, E \vdash s; s' \rightarrow v', (f; f'), C' \\ \text{and } \mathcal{E}, \ell \vdash s; s' : \tau', \mathcal{E}', (\sigma; \sigma')$$

By the semantics, this requires:

$$C_0, E \vdash s \rightarrow v, f, C \text{ and } C, E \vdash s' : v', f', C' \\ \text{and } \mathcal{E}, \ell \vdash s : \tau, \mathcal{E}_1, \sigma \text{ and } \mathcal{E}_1, \ell \vdash s' : \tau', \mathcal{E}', \sigma'$$

By induction on  $s$ , there exists a store  $\mathcal{S}_1$  such that  $\mathcal{C}_0 : \mathcal{S} \models \mathcal{C} : \mathcal{S}_1$  verifying:

$$\mathcal{S}_1 \models f : \sigma \text{ and } \mathcal{C} : \mathcal{S}_1 \models v' : \tau' \text{ and } \mathcal{C} : \mathcal{S}_1 \models E : \mathcal{E}_1$$

By induction on  $s'$ , there exists a store  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S}_1 \sqsubseteq \mathcal{C}' : \mathcal{S}'$  verifying:

$$\mathcal{S}' \models f' : \sigma' \text{ and } \mathcal{C}' : \mathcal{S}' \models v' : \tau' \text{ and } \mathcal{C}' : \mathcal{S}' \models E : \mathcal{E}'$$

Thus, we conclude:

$$\mathcal{S}' \models f; f' : \sigma; \sigma' \text{ and } \mathcal{C}' : \mathcal{S}' \models v' : \tau' \text{ and } \mathcal{C}' : \mathcal{S}' \models E : \mathcal{E}'$$

- **Case of (If-T):** By hypothesis, we have:

$$\begin{aligned} & \mathcal{C}_0 : \mathcal{S} \models E : \mathcal{E} \\ & \text{and } \mathcal{C}_0, E \vdash \text{if } e \text{ then } s' \text{ else } s'' \rightarrow v', (f; f'), \mathcal{C}' \\ & \text{and } \mathcal{E}, \ell \vdash \text{if } e \text{ then } s' \text{ else } s'' : \text{if}(\tau', \tau''), \mathcal{E}' \mathbb{M} \mathcal{E}'', (\sigma; \text{if}(\sigma', \sigma'')) \end{aligned}$$

By the semantics, this requires:

$$\begin{aligned} & \mathcal{C}_0, E \vdash e \rightarrow v, f, \mathcal{C} \text{ and } v \neq 0 \text{ and } \mathcal{C}, E \vdash s' : v', f', \mathcal{C}' \\ & \text{and } \mathcal{E}, \ell \vdash e : \text{int}_\mu, \sigma \text{ and } \mathcal{E}, \ell \vdash s' : \tau', \mathcal{E}', \sigma' \text{ and } \mathcal{E}, \ell \vdash s'' : \tau'', \mathcal{E}'', \sigma'' \end{aligned}$$

By induction on  $e$ , there exists a store model  $\mathcal{S}_1$  such that  $\mathcal{C}_0 : \mathcal{S} \sqsubseteq \mathcal{C} : \mathcal{S}_1$  and  $\mathcal{C} : \mathcal{S}_1 \models f : \sigma$ .

By induction on  $s'$ , there exists a store model  $\mathcal{S}'$  such that  $\mathcal{C} : \mathcal{S}_1 \sqsubseteq \mathcal{C}' : \mathcal{S}'$

$$\mathcal{S}' \models f' : \sigma' \text{ and } \mathcal{C}' : \mathcal{S}' \models v' : \tau' \text{ and } \mathcal{C}' : \mathcal{S}' \models E : \mathcal{E}'$$

By definition, we have:

$$\mathcal{C}' : \mathcal{S}' \models f' : \text{if}(\sigma', \sigma'') \text{ and } \mathcal{C}' : \mathcal{S}' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}''$$

Thus, we conclude:

$$\mathcal{C}' : \mathcal{S}' \models f; f' : \sigma; \text{if}(\sigma', \sigma'') \text{ and } \mathcal{C}' : \mathcal{S}' \models v' : \tau' \text{ and } \mathcal{C}' : \mathcal{S}' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}''$$

- **Case of (If-F):** By hypothesis, we have:

$$\begin{aligned} & \mathcal{C}_0 : \mathcal{S} \models E : \mathcal{E} \\ & \text{and } \mathcal{C}_0, E \vdash \text{if } e \text{ then } s' \text{ else } s'' \rightarrow v'', (f; f''), \mathcal{C}'' \\ & \text{and } \mathcal{E}, \ell \vdash \text{if } e \text{ then } s' \text{ else } s'' : \text{if}(\tau', \tau''), \mathcal{E}' \mathbb{M} \mathcal{E}'', (\sigma; \text{if}(\sigma', \sigma'')) \end{aligned}$$

By the semantics, this requires:

$$C_0, E \vdash e \rightarrow v, f, C \text{ and } v = 0 \text{ and } C, E \vdash s'' : v'', f'', C'' \\ \text{and } \mathcal{E}, \ell \vdash e : \text{int}_\mu, \sigma \text{ and } \mathcal{E}, \ell \vdash s' : \tau', \mathcal{E}', \sigma' \text{ and } \mathcal{E}, \ell \vdash s'' : \tau'', \mathcal{E}'', \sigma''$$

By induction on  $e$ , there exists a store model  $\mathcal{S}_1$  such that  $C_0 : \mathcal{S} \sqsubseteq C : \mathcal{S}_1$  verifying  $\mathcal{S}_1 \models f : \sigma$ .

By induction on  $s''$ , there exists a store model  $\mathcal{S}''$  such that  $C : \mathcal{S}_1 \sqsubseteq C'' : \mathcal{S}''$  verifying:

$$\mathcal{S}'' \models f'' : \sigma'' \text{ and } C'' : \mathcal{S}'' \models v'' : \tau'' \text{ and } C'' : \mathcal{S}'' \models E : \mathcal{E}''$$

By definition, we have:

$$\mathcal{S}'' \models f'' : \text{if}(\sigma', \sigma'') \text{ and } C'' : \mathcal{S}'' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}''$$

Thus, we conclude:

$$\mathcal{S}'' \models f; f'' : \sigma; \text{if}(\sigma', \sigma'') \text{ and } C'' : \mathcal{S}'' \models v'' : \tau'' \text{ and } C'' : \mathcal{S}'' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}''$$

- **Case of (While-T):** By hypothesis, we have:

$$C_0 : \mathcal{S} \models E : \mathcal{E} \text{ and } C_0, E \vdash \text{while } e \text{ do } s \rightarrow v', f', C' \\ \text{and } \mathcal{E}, \ell \vdash \text{while } e \text{ do } s : \text{if}(\tau', \text{void}), \mathcal{E}' \mathbb{M} \mathcal{E}, \text{if}((\sigma; \sigma'), \sigma)$$

By the semantics, this requires:

$$C_0, E \vdash e \rightarrow v, f, C \text{ and } v \neq 0 \text{ and } C, E \vdash s : v', f', C' \\ \text{and } \mathcal{E}, \ell \vdash e : \text{int}_\mu, \sigma \text{ and } \mathcal{E}, \ell \vdash s : \tau', \mathcal{E}', \sigma'$$

By induction on  $e$ , there exists a store  $\mathcal{S}_1$  such that  $C_0 : \mathcal{S} \sqsubseteq C : \mathcal{S}_1$  and  $\mathcal{S}_1 \models f : \sigma$ .

By induction on  $s$ , there exists a store model  $\mathcal{S}'$  such that  $C : \mathcal{S}_1 \sqsubseteq C' : \mathcal{S}'$  verifying:

$$\mathcal{S}' \models f' : \sigma' \text{ and } C' : \mathcal{S}' \models v' : \tau' \text{ and } C' : \mathcal{S}' \models E : \mathcal{E}'$$

By definition, we have:

$$\mathcal{S}' \models f; f' : \sigma; \sigma' \text{ and } \mathcal{S}' \models f; f' : \text{if}((\sigma; \sigma'), \sigma) \text{ and } C' : \mathcal{S}' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}$$

Thus, we conclude:

$$\mathcal{S}' \models f; f' : \text{if}((\sigma; \sigma'), \sigma) \text{ and } C' : \mathcal{S}' \models v' : \tau' \text{ and } C' : \mathcal{S}' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}$$

- **Case of (While-F):** By hypothesis, we have:



$$\begin{aligned} & C : S \models E : \mathcal{E} \text{ and } C, E \vdash \text{while } e \text{ do } s \rightarrow \text{unit}, f, C' \\ & \text{and } \mathcal{E}, \ell \vdash \text{while } e \text{ do } s : \text{if}(\tau', \text{void}), \mathcal{E}' \mathbb{M} \mathcal{E}, \text{if}((\sigma; \sigma'), \sigma) \end{aligned}$$

By the semantics, this requires:

$$\begin{aligned} & C, E \vdash e \rightarrow v, f, C' \text{ and } v = 0 \\ & \text{and } \mathcal{E}, \ell \vdash e : \text{int}_\mu, \sigma \text{ and } \mathcal{E}, \ell \vdash s : \tau', \mathcal{E}', \sigma' \end{aligned}$$

By induction on  $e$ , there exists a store model  $S'$  such that  $C : S \sqsubseteq C' : S'$  verifying  $S' \models f : \sigma$ . By transitivity of  $\models$ , we have  $C' : S' \models E : \mathcal{E}$ . By definition, we have:

$$S' \models f : \text{if}((\sigma; \sigma'), \sigma) \text{ and } C' : S' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}$$

Thus, we conclude:

$$S' \models f : \text{if}((\sigma; \sigma'), \sigma) \text{ and } C' : S' \models \text{unit} : \text{if}(\tau', \text{void}) \text{ and } C' : S' \models E : \mathcal{E}' \mathbb{M} \mathcal{E}.$$

■

## 5.4 Soundness of Static Analysis

We have shown in the previous section that our static semantics and dynamic semantics are consistently related. In this section, we define the soundness property of our static analysis. Then, based on the consistency results of our static and dynamic semantics, we establish the soundness proof to demonstrate that our analysis does not suffer false negatives. To this end, we enrich the operational semantics with error rules that capture the undefined behaviours of memory operations. These unsafe behaviours result in dynamic errors that are predicted by our static checks as stated in our Soundness Theorem 5.4.1. Notice that our operational semantics does not consider modification to memory layouts during cast operations. Thus, it does not capture runtime errors related to type conversions.

**Theorem 5.4.1 (Static Analysis Soundness)** *Let  $E$  be an environment and  $\mathcal{E}$  its type. Let  $C : S$  be a typed store such that  $C : S \models E : \mathcal{E}$ . Let  $e$  be an expression such that  $\mathcal{E}, \ell \vdash e : \tau, \sigma$  and  $C, E \vdash e \rightarrow v, f, C'$  and  $C : S \models v : \tau$  and  $C : S \models f : \sigma$ :*

- *If  $C, E \vdash *e \rightarrow \text{error}$  then  $\text{drfChk}(\tau) = \text{false}$ .*
- *If  $C, E \vdash \text{free}(e) \rightarrow \text{error}$  then  $\text{freeChk}(\tau, \sigma) = \text{false}$ .*
- *If  $C, E \vdash e.\varphi \rightarrow \text{error}$  then  $\text{fldChk}(\tau, \varphi) = \text{false}$ .*

**Proof of Static Analysis Soundness 5.4.1** To establish the soundness proof, we define an error rule for each memory operation that has an undefined behaviour as stated by the ANSI C standard [75].

- **Unsafe pointer dereference:** In the dynamic semantics a pointer dereferencing error is captured by the following rule:

$$\frac{C, E \vdash e \rightarrow v, f, C' \quad v \notin \text{Ref}}{C, E \vdash *e \rightarrow \text{error}}$$

In the static semantics, the (Deref) rule is as follows:

$$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma \quad \bar{\tau} = \text{ref}(\_) \quad \text{drfChk}(\tau) \quad \rho = \text{regionOf}(\tau) \quad \tau' = \text{strTypeOf}(\tau)}{\mathcal{E}, \ell \vdash *e : \tau', (\sigma; \text{read}(\rho, \tau', \ell))}$$

The dereference of pointer type  $\tau$  is guarded by the following static check:

$$\text{drfChk}(\tau) = (\bar{\tau} = \text{ref}(\kappa)) \wedge (\kappa \neq \text{void}) \wedge (\text{hostOf}(\tau) \notin \{[\text{wild}], [\text{dangling}], [\text{arith}]\})$$

We consider the possible values of expression  $e$  that lead to a runtime error:

– If  $v \notin \text{Ref}$  and  $v = \text{undef}$ , we have  $\mathcal{C} : \mathcal{S} \models \text{undef} : \tau$ , thus type  $\tau$  is such that:

$$(\bar{\tau} \in \{\text{void}, \text{ref}(\text{void})\}) \vee ((\text{hostOf}(\tau) \subseteq \{\text{wild}, [\text{dangling}], [\text{arith}]\}))$$

Thus, we conclude that  $\text{drfChk}(\tau) = \text{false}$ .

– If  $v \notin \text{Ref}$  and  $v \neq \text{undef}$ , we have  $\mathcal{C} : \mathcal{S} \models v : \tau$  and this implies  $\bar{\tau}$  not of the form  $\text{ref}(\kappa)$ . Thus, we conclude that  $\text{drfChk}(\tau) = \text{false}$ .

- **Unsafe pointer deallocation:** In the dynamic semantics a pointer deallocation error is captured by the following rule:

$$\frac{\mathcal{C}, E \vdash e \rightarrow v, f, \mathcal{C}' \quad ((v \notin \text{Ref}) \vee (\text{stack}(v) \in f) \vee (\text{free}(v) \in f))}{\mathcal{C}, E \vdash \text{free}(e) \rightarrow \text{error}}$$

In the static semantics, the (Free) rule is as follows:

$$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma \quad \bar{\tau} = \text{ref}(\_) \quad \text{freeChk}(\tau, \sigma) \quad \rho = \text{regionOf}(\tau) \quad \mathcal{E}' = \text{updEnv}(\mathcal{E}, \text{free}(e), \tau)}{\mathcal{E}, \ell \vdash \text{free}(e) : \text{void}, \mathcal{E}', (\sigma; \text{dealloc}(\rho, \ell))}$$

The memory deallocation of pointer type  $\tau$  is guarded by the following static check:

$$\text{freeChk}(\tau, \sigma) = (\bar{\tau} = \text{ref}(\kappa)) \wedge (\kappa \neq \text{void}) \wedge (\text{hostOf}(\tau) = [\&\tau']) \wedge (\text{stack}(\rho, \ell) \notin \sigma)$$

By consistency hypothesis, we have  $\mathcal{C} : \mathcal{S} \models v : \tau$  and  $\mathcal{C} : \mathcal{S} \models f : \sigma$ . We cover the three cases that result in a runtime error:

– If  $v \notin \text{Ref}$  and  $v = \text{undef}$ , by definition of the consistency relation  $\mathcal{C} : \mathcal{S} \models v : \tau$ , we have:

$$(\bar{\tau} \in \{\text{void}, \text{ref}(\text{void})\}) \vee ((\text{hostOf}(\tau) \subseteq \{\text{wild}, \text{dangling}, \text{arith}\}))$$

Thus, we conclude that  $\text{freeChk}(\tau, \sigma) = \text{false}$ .

– If  $v \notin \text{Ref}$  and  $v \neq \text{undef}$ , by definition of the consistency relation  $\mathcal{C}$  :

$\mathcal{S} \models v : \tau$ , we have  $\bar{\tau}$  not of the form  $\text{ref}(\kappa)$ . Thus, we conclude that

$\text{freeChk}(\tau, \sigma) = \text{false}$ .

– If  $\text{stack}(v) \in f$ , by consistency definition we have:

$$\mathcal{S}(v) = (\rho, \text{strTypeOf}(\tau)) \text{ and } \text{stack}(\rho, \ell) \in \sigma$$

Thus, we conclude that  $\text{freeChk}(\tau, \sigma) = \text{false}$ .

- **Unsafe field access:** In the dynamic semantics a field access error is captured

by the following rule:

$$\frac{\mathcal{C}, E \vdash e \rightarrow v, f, \mathcal{C}' \quad ((v = \text{loc}@ \langle v_i \rangle_{i=1..n} \wedge \mathcal{C}(\text{loc} + \text{offset}(\varphi)) \neq v_i) \vee (v \text{ not of the form } \text{loc}@ \langle \rangle))}{\mathcal{C}, E \vdash e.\varphi \rightarrow \text{error}}$$

In the static semantics, the (Field) rule is as follows:

$$\frac{\mathcal{E}, \ell \vdash e : \tau, \sigma \quad \bar{\tau} = \text{struct}\{\_ \} \quad \text{fldChk}(\tau, \varphi) \quad \tau' = \text{fldType}(\tau, \varphi) \quad \rho = \text{addressOf}(e.\varphi, \mathcal{E})}{\mathcal{E}, \ell \vdash e.\varphi : \tau', \sigma}$$

The field dereference operation  $e.\varphi$  is guarded by the following static check:

$$\text{fldChk}(\tau, \varphi) = \varphi \in \text{fldList}(\tau)$$

– if  $v = \text{loc}@ \langle v_i \rangle_{i=1..n}$ , by definition of the consistency relation  $\mathcal{C} : \mathcal{S} \models$

$\text{loc}@ \langle v_i \rangle_{i=1..n} : \tau$ , we have:

$$\forall \varphi_i \in \text{fldList}(\tau), \mathcal{C}(\text{loc} + \text{offset}(\varphi_i)) = v_i$$

Since  $\mathcal{C}(\text{loc} + \text{offset}(\varphi)) \neq v_i$  then  $\varphi \notin \text{fldList}(\tau)$ .

Thus, we conclude that  $\text{fldChk}(\tau, \varphi) = \text{false}$

– if  $v$  not of the form  $\text{loc}@(\_)$ , by definition of the consistency relation

$\mathcal{C} : \mathcal{S} \models v : \tau$ , we have  $\bar{\tau}$  not of the form  $\text{struct}\{\}$ .

Thus, we conclude that  $\text{fldChk}(\tau, \varphi) = \text{false}$ . ■

## 5.5 Guiding Code Instrumentation

As for all static techniques, our conservative type analysis generates false positives and has undecidability issues. In this section, we show that our static approach can be supplemented with a dynamic counterpart to increase the overall precision.

### 5.5.1 Static Analysis Limitations

Our conservative type and effect analysis carries out an exhaustive coverage of programs' execution traces. From its path-insensitivity, it loses precision by assuming that all paths are feasible, thus it tends to generate false positives. Supplementing our static analysis with runtime checks presents the advantages of spotting feasible paths and eliminating false positives. Nevertheless, runtime checks induce performance overhead on the analyzed program. As such, we should reduce resorting to runtime verification to a bare minimum. To this end, we classify the results of our static checks into three categories:

- A static check is successful for a given memory operation when all execution traces leading to that operation are safe.
- A static check is unsuccessful for a given memory operation when all execution traces leading to that operation are unsafe.
- A static check is undecidable for a given memory operation when at least one execution trace leading to that operation is unsafe. As this trace may be unfeasible, we resort to dynamic analysis to spot the actual runtime error if any.

From this classification of static checks, we deduce that dynamic analysis is used to handle statically undecidable program operations exclusively. We define an effect-based approach to interface static analysis with a dynamic counterpart. The effects collected during the type analysis provide a tree-based model of a program that captures control-flow and alias information. We extract from this effect model what we call a *dunno point* that characterizes a statically undecidable operation as detailed in what follows.

### 5.5.2 Static *Dunno Points*

In Table 5.5.2, we define a dunno point *dunno* as a three-tuple  $\langle chkTag, e, \pi \rangle$  where *chkTag* is a tag that describes the needed runtime check, *e* is the expression to check, and  $\pi$  represents a common signature of all execution traces that ends with the suspicious operation.

<sup>1</sup> A program may have a large number of error traces, especially when it contains

---

**Table 15** *Dunno Points* to guide code instrumentation

---

<i>dunnoPoint</i>	$\ni$	<i>dunno</i>	$::=$	<i>checkTag</i> $\times$ <i>Expression</i> $\times$ <i>pathSig</i>
<i>checkTag</i>	$\ni$	<i>chkTag</i>	$::=$	<i>Wild</i>
				<i>Dangling</i>
				<i>Bounds</i>
				<i>DblFree</i>
				<i>StackFree</i>
				<i>InitRhs</i>
				<i>FldStr</i>
<i>pathSig</i>	$\ni$	$\pi$	$::=$	$\emptyset$
				$\mu$
				$(\ell, true) \Rightarrow \pi$
				$(\ell, false) \Rightarrow \pi$
				$\pi \Rightarrow \mu$

$\mu \in \{alloc(\rho, \ell), dealloc(\rho, \ell), read(\rho, \tau, \ell), assign(\rho, \tau, \ell)\}$

---

loops. Extracting all these error traces is cumbersome and useless since many of them are similar and lead to the same error. In order to reduce the number of reported traces, we define the domain *pathSig*, ranged over by  $\pi$ , that gives a concise characteristic of error traces. It identifies the specific branches and program points that define the bad sequencing of memory operations. To prevent our analysis from being trapped in an infinite loop, we set an arbitrary number of iterations for all loops in the analyzed program.

The element  $(\ell, true)$  follows the true branch at the branching point  $\ell$ , whereas  $(\ell, false)$  follows the false branch. The notation  $(\ell, true) \Rightarrow (\ell', \_)$  denotes all paths that reach program point  $\ell'$  through the true branch at location  $\ell$ . On the other hand,  $(\ell, false) \Rightarrow (\ell', \_)$  represents all paths that reach location  $\ell'$  through the false branch at program point  $\ell$ . The last element of a signature  $\pi$  is an effect element  $\mu$

that corresponds to the vulnerable operation. Expanding a path signature  $\pi$  yields all execution traces that lead to the same suspected error.

In addition to the error traces, a dunno point indicates the needed runtime check by specifying a tag from the domain *checkTag*. As such, our analysis generates the following set of dunno points when facing undecidability:

- $\langle Wild, e, \pi \Rightarrow read(\_, \ell) \rangle$ : check if pointer  $e$  is null or uninitialized before dereferencing at program point  $\ell$  for execution paths corresponding to signature  $\pi$ .
- $\langle Dangling, e, \pi \Rightarrow read(\_, \ell) \rangle$ : check if pointer  $e$  is dangling before dereferencing at program point  $\ell$  for execution paths corresponding to signature  $\pi$ .
- $\langle Bounds, e, \pi \Rightarrow read(\_, \ell) \rangle$ : check if pointer  $e$  is out-of-bounds before dereferencing at program point  $\ell$  for execution paths corresponding to signature  $\pi$ .
- $\langle DblFree, e, \pi \Rightarrow dealloc(\_, \ell) \rangle$ : check if pointer  $e$  is not dangling before freeing at program point  $\ell$  for execution paths corresponding to signature  $\pi$ .
- $\langle StackFree, e, \pi \Rightarrow dealloc(\_, \ell) \rangle$ : check if pointer  $e$  refers to a dynamically allocated memory before freeing at program point  $\ell$  for execution paths corresponding to signature  $\pi$ .
- $\langle InitRhs, e, \pi \Rightarrow assign(\_, \ell) \rangle$ : check if the right-hand-side operand  $e$  of the assignment at program point  $\ell$  is initialized for execution paths corresponding to signature  $\pi$ .



---

**Table 16** Operator  $\oplus$  for combining *dunno* points

---

$\oplus$	$(0, \textit{dunno}')$	$(1/2, \textit{dunno}')$	$(1, \emptyset)$
$(0, \textit{dunno})$	$(0, \textit{dunno} \cup \textit{dunno}')$	$(1/2, \textit{dunno} \cup \textit{dunno}')$	$(1/2, \textit{dunno})$
$(1/2, \textit{dunno})$	$(1/2, \textit{dunno} \cup \textit{dunno}')$	$(1/2, \textit{dunno} \cup \textit{dunno}')$	$(1/2, \textit{dunno})$
$(1, \emptyset)$	$(1/2, \textit{dunno}')$	$(1/2, \textit{dunno}')$	$(1, \emptyset)$

---

- $\langle \textit{FldStr}, e, \varphi, \pi \Rightarrow \textit{read}(\_, \ell) \rangle$ : check if structure  $e$  has a field  $\varphi$  before access at program point  $\ell$  for execution paths corresponding to signature  $\pi$ .

### 5.5.3 Locating Instrumentation Points

Now that we have defined *dunno* points, we revisit the output of our static checks in order to generate more expressive results as defined hereafter:

- $(1, \emptyset)$  when successful.
- $(0, \textit{dunno})$  when failed, the element *dunno* is used to extract the execution trace leading to an error.
- $(1/2, \textit{dunno})$  when facing undecidability, the element *dunno* is used to extract the execution traces that should be instrumented with runtime checks.

As specified previously, static undecidability occurs when traces leading to the same operation on a given expression diverge on the safety of that operation. In other words, the expression in question has different types enclosed in a type construct  $\textit{if}(\_, \_)$  where some types pass the static check, while others fail. In Table 16, we define the operator  $\oplus$  that combines the results of a static check applied on an  $\textit{if}(\_, \_)$

---

**Algorithm 7** Revised static checks for dunno point generation
 

---

```

Function drfChk( $e, \tau, \pi$ ) =
  case  $\tau$  of
     $if_{\ell}(\tau', \tau'')$        $\Rightarrow$   $drfChk(e, \tau', \pi \rightarrow (\ell, true)) \oplus drfChk(e, \tau'', \pi \rightarrow (\ell, false))$ 
     $ref_{\rho}(\_)[dangling]$      $\Rightarrow$   $(0, \langle Dangling, e, \pi \rightarrow read(\rho, \ell) \rangle)$ 
     $ref_{\rho}(\_)[wild]$          $\Rightarrow$   $(0, \langle Wild, e, \pi \rightarrow read(\rho, \ell) \rangle)$ 
     $ref_{\rho}(\_)[arith]$         $\Rightarrow$   $(0, \langle Bounds, e, \pi \rightarrow read(\rho, \ell) \rangle)$ 
    else                     $\Rightarrow$   $(1, \emptyset)$ 
  end
Function freeChk( $\tau, \sigma, \pi$ ) =
  case  $\tau$  of
     $if_{\ell}(\tau', \tau'')$        $\Rightarrow$   $freeChk(\tau, \sigma, \ell \rightarrow \pi) \oplus freeChk(e, \tau', \ell \rightarrow \pi)$ 
     $ref_{\rho}(\_)[dangling]$      $\Rightarrow$   $(0, \langle DblFree, e, \pi \rangle)$ 
     $ref_{\rho}(\_)[wild]$          $\Rightarrow$   $(0, \langle UnallocFree, e, \pi \rangle)$ 
     $ref_{\rho}(\_)[arith]$         $\Rightarrow$   $(0, \langle Bounds, e, \pi \rangle)$ 
    else                     $\Rightarrow$  if  $(stack(\rho, \_) \in \sigma)$  then
       $(0, \langle UnallocFree, e, \pi \rangle)$ 
    else
       $(1, \emptyset)$ 
  end
Function asgnChk( $e, \tau, \tau', \pi$ ) =
  case  $\tau'$  of
     $if_{\ell}(\tau_1, \tau_2)$        $\Rightarrow$   $asgnChk(e, \tau, \tau_1, \ell \rightarrow \pi) \oplus asgnChk(e, \tau, \tau_2, \ell \rightarrow \pi)$ 
     $ref_{\rho}(\_)\eta | int_{\eta}$      $\Rightarrow$  if  $(\eta \notin \{dangling, wild, arith\})$  then
       $(0, \langle InitRv, e, \pi \rangle)$ 
    else
       $(1, \emptyset)$ 
  end
Function fldChk( $e.\varphi, \tau, \pi$ ) =
  case  $\tau$  of
     $if_{\ell}(\tau_1, \tau_2)$        $\Rightarrow$   $fldChk(e.\varphi, \tau_1, \ell \rightarrow \pi) \oplus fldChk(e.\varphi, \tau_2, \ell \rightarrow \pi)$ 
     $\tau'$                      $\Rightarrow$  if  $(\varphi \notin fldList(\tau'))$  then
       $(0, \langle fldStr, e.\varphi, \pi \rangle)$ 
    else
       $(1, \emptyset)$ 
  end

```

---

type. The combination returns a global result for the static check. For a given type  $\tau = \text{if}(\tau', \tau'')$ , if the check result for  $\tau'$  is  $(1, \emptyset)$  and  $(0, \text{dunno})$  for  $\tau''$  (or vice-versa), then the overall result for  $\tau$  is undecidable and equal to  $(1/2, \text{dunno})$ . Notice that an undecidable output prevails over all other outputs. The operator  $\oplus$  performs the union of dunno points since each dunno point is related to a different execution trace signature. The algorithms of the static checks defined in the previous chapter are revised in Algorithm 7 in order to derive dunno points. Notice that we decorate the  $\text{if}_\ell(\_, \_)$  type with a program point annotation  $\ell$  that captures the location of the branching statement. The program point annotation is used to extract error traces for dunno points.

## 5.6 Extending GCC

We have chosen to implement our safety verification approach as an extension of the GCC compiler. Our implementation is based on the GCC core distribution version 4.2.0. We made this choice for the following reasons:

- GCC is the defacto compiler of C programs, thus integrating our safety verification within the GCC compiler increases its usability during software development processes. Our safety verification is launched during the compilation process by simply setting its corresponding flag when invoking GCC.
- Starting from version 4.0, the GCC compiler is based on the Tree-SSA framework for the development of high-level code optimization techniques and static

---

**Table 17** Experimental results illustrating the performance of our approach

---

File	LOC (C code)	Compile without checks (secs)	Compile with checks (secs)	Slowdown factor
openssh-5.0p1	46.33K	118	611	5.17
openssl-0.9.8j	187.101K	358	1200	3.35
Filesystem Linux-2.6.26.6	12.153K	90	338	3.75

---

analysis tools [97]. The Tree-SSA framework provides an easy access to control-flow, data-flow, and type information, thus it facilitates the implementation of our static analysis. We also took advantage of the GIMPLE intermediate representation language provided by Tree-SSA. GIMPLE preserves source-level information about the code but simplifies complex constructs (e.g., loops are mapped to if and goto statements).

To enable our static analysis, we pass the `-fipa-annot-inference` command-line option to the extended GCC compiler. The normal compilation process of the compiler remains intact. When memory and type errors are detected, our type analysis pass generates warnings. We analyzed a set of real software such as `openssh-5.0p1`, `openssl-0.9.8j`, and a part of the `Linux-2.6.26.6` filesystem in order to demonstrate the scalability of our prototype. Table 17 gives the overhead on the compilation time imposed by our safety analysis. The measurements were made on a 1GHz Intel, 1GB Linux machine, using the GCC-4.2.1 compiler with `-O` optimization level. During the experimentation, we activated some of our safety checks in order to detect: free of dangling pointers, free of unallocated pointers, dereference of dangling pointers, and bad cast from integer to pointer.

We detected a bad cast operation from integer to pointer in the Linux kernel function `vmsplice_to_user()` (`fs/splice.c`). It actually corresponds to the well known `vmsplice` local root exploit (BID: 27704) that takes advantage of a pointer copied from the user space by the kernel function `get_user`. According to the Linux system call specifications, `get_user` is used to get an integer value from the user space [22]. In the `vmsplice_to_user()` function, the usage of `get_user` is not conform to its specification since it copies a pointer value from the user space instead of an integer value. Moreover, the address referred to by the user space pointer is never validated before being used.

Listing 5.1: Vulnerable Linux Function `vmsplice_to_user()`

```
error = get_user(base, &iov->iov_base);
/* ... */
if (unlikely(!base)) {
    error = -EFAULT;
    break;
}
/* ... */
sd.u.userptr = base;
/* ... */
size = __splice_from_pipe(pipe, &sd, pipe_to_user);
```

Listing 5.2: GIMPLE Representation of `vmsplice_to_user()`

```
long unsigned int __val_gu;
/* ... */
void * base;
/* ... */
D.25866 = &iov->iov_base;
__asm__ __volatile__ ("call __get_user_4" : "=a" __ret_gu, "=d" __val_gu : "0" D.25866);
base = (void *) __val_gu;
D.25845 = __ret_gu;
error = D.25845;
```

Listing 5.1 provides an extract of the vulnerable Linux code and a relevant snippet of its GIMPLE representation in Listing 5.2. The latter replaces the call to `get_user` with its inline assembly code where: `__get_user_4` is the invoked assembly function, `__ret_gu` is the return value of the call, `__val_gu` is the integer value copied from the user space, and `D.25866` is a temporary variable generated by GIMPLE corresponding

to the user space address to copy from. From the GIMPLE code, we can detect that the integer value `__val_gu` is cast to void pointer and assigned to the void pointer base. This cast operation fails our safety check `castChk` that entails that only integers previously derived from pointers can be cast back to pointer type. We are not aware of any static analysis tool that has discovered this error before being exploited. This experimentation demonstrates the scalability of our prototype and its potential in detecting real errors.

## 5.7 Conclusion

In this chapter, we defined an operational semantics of our imperative language that complies with the ANSI C standard. We enriched our operational semantics with error rules where undefined behaviours related to our targeted set of unsafe memory operations are evaluated to runtime errors. We proved the consistency between our static semantics and dynamic semantics. The consistency results were used to show that our static safety checks conservatively detect all occurrences of their targeted memory errors. Moreover, we defined an effect-based approach that allows our static analysis to be supplemented with a dynamic counterpart in order to overcome the inevitable static undecidability issues. We also prototyped our static analysis as an extension of the GCC compiler. The experimental results presented in this chapter demonstrates the scalability and the efficiency of our approach.

# Chapter 6

## Automatic Security Verification

### 6.1 Introduction

In this chapter, we define our automated approach that combines static analysis and model-checking for the security verification of source code. As previously detailed in Chapter 2, software model-checking [15, 32, 68] is more efficient than static analysis in specifying and verifying system-specific security properties. Nevertheless, the state explosion problem is the main issue of software model-checking. The state space to explore grows exponentially with respect to the size of the analyzed program abstractions [38]. This problem limits the applicability and the usability of model-checking for large software verification. Abstraction is a well-known and established technique to cope with the state explosion problem. Thus, the model-checking challenge is the generation of a scalable and concise abstraction of programs.

The core idea of our approach is to utilize static analysis for the automatic generation and optimization of model-checkable program abstractions. As a result, our approach can model-check large software against customized system-specific security properties. Since we target open source software, we base our approach on GCC the defacto open source compiler to benefit from its language-independent and platform-independent GIMPLE intermediate representation of source code. For the verification process, we use the Moped model-checker for pushdown systems [107] that comes with a procedural input language called Remopla. As such, we automatically extract, from GIMPLE representations, program behaviours that are relevant to the considered security properties and serialize them into Remopla representations of Moped. In addition, we enrich program abstractions with a Remopla constructs that compute and capture data dependencies between program expressions. Hence, we are able to detect insidious errors that involve variable aliasing and function parameter passing. Security properties and program Remopla model are input to Moped in order to detect security violations and provide witness paths leading to them.

The chapter is organized as follows: The software components and the overall architecture of our approach are outlined in Section 6.2. Specification of security properties as finite state automata is detailed in Section 6.3. The generation of program Remopla models is presented in Section 6.4. Section 6.5 is dedicated to our approach for handling data dependencies between program expressions. Our static analysis based technique for safe and concise abstraction of program models is presented in Section 6.6. We draw conclusion of this chapter in Section 6.7.



## 6.2 Approach Overview

In this section, we outline an overview of our security verification environment. First, we give a short introduction of the software components that provide the basis for our approach. Then, we describe the architecture depicted in Figure 3 that integrates these components and the modus operandi of our approach.

### 6.2.1 Tree-SSA Framework

Our ultimate goal is to provide a security verification environment for open source software. To achieve our objective, the GCC compiler fulfills our requirement for the multi-language support. Since the last decades, the GCC compiler is considered as the defacto compiler for open source. Moreover, starting from version 4.0, the GCC main-line includes the Tree-SSA framework for code optimization and static analysis [97]. It provides the language and platform independent GIMPLE tree representation of source code that facilitates the analysis of the compiler intermediate code. GIMPLE simplifies complex structures such as flattening loops into `if\then\else` and `goto` statements. It also converts expressions into a 3-address code of SSA form [44], using temporary variables to hold intermediate values. Working with GIMPLE representation allows our analysis to be focused more on data modifications and control flow information instead of putting effort into analyzing complex language constructs.

## 6.2.2 Moped Model-Checker

Moped is a model-checking tool for pushdown-systems based on the algorithms defined in [50]. The first version performs a combined linear temporal logic and reachability model-checking. Since 2005, version 2 of Moped comes with a new modeling language called Remopla, which is a C-like language and features explicit procedures [3]. The key feature of the new version is its implementation of abstraction refinement using Binary-Decision Diagrams (BDD) that enhances the performance of Moped [51]. For now, Moped version 2 only performs reachability analysis.

**Table 18** Remopla language constructs

Remopla Constructs	
Data types	bool, int, struct, enum, void
Statements	skip, break, goto, return
Conditional	if, else, fi
Loop control	do, od
Special values	undef, true, false, DEFAULT_INT_BITS
Others	define, init, module

**Listing 6.1:** sample C code with Remopla Representation

<pre>int f(int x) {   if(x &lt; 5)     x = x + 1;   else     x = x - 1;   return x; } void main () {   int i;   ...   i = f(i); }</pre>	<pre>define DEFAULT_INT_BITS 5 init main ;  module f(int x) {   if   :: x &lt; 5 -&gt; L1: x = x + 1;   :: else -&gt; L2: x = x - 1;   fi;   return x; } module main() {   int i;   ...   i = f(i);   ... }</pre>
(a) Sample C Code	(b) Remopla Code

Our software verification environment is based on the second version of Moped in order to benefit from the expressiveness of the Remopla language. A Remopla model of a given program consists of a set of module definitions characterizing the behaviour of the model. A Remopla module implements the concept of functions and procedures with a body that comprises a sequence of Remopla statements. Table 18 summarizes the basic Remopla constructs. To build Remopla models of large C software, we automatically serialize GIMPLE representation into Remopla code. In Listing 6.1, we show a sample C code with its Remopla model. Each Remopla statement can be assigned a label that is used for reachability analysis. To verify whether a given label is reachable, Moped is invoked with the following command line: `#moped <Remopla_file_name> <label>`. In the given Remopla sample, label L1 is reached when  $x$  is less than 5, whereas label L2 is reached when  $x$  is greater than 5. The `DEFAULT_INT_BITS` set to 5 indicates that the value of variable  $x$  can range from 0 to  $2^5 - 1$ .

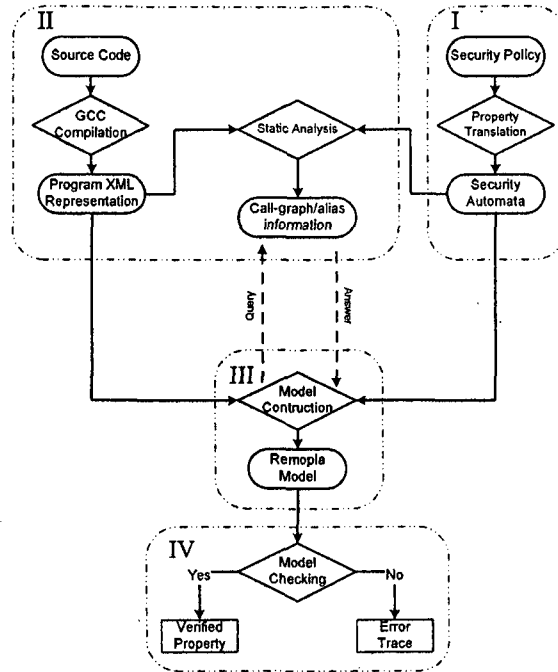
### 6.2.3 Architecture

Figure 3 depicts the architecture of our security verification environment. The security verification of programs is carried out through different phases including security property specification, program model extraction, and property model-checking. In the following paragraphs, we informally describe the input, the output, and the tasks of each of these phases.

---

**Figure 3** Security verification framework

---



---

### Phase1: Security Property Specification

- Input: Security properties.
- Output: Remopla automata of security properties.

The first step of our verification process requires the definition of security properties describing what not to do for the purpose of building secure code. We provide programmers with a tool in order to graphically characterize the security rules that a program should obey. Each property is specified as a finite state automaton where the nodes represent program states and the transitions match program actions. Final states of automata are risky states that should never be reached. To ease the property

specification, our tool supports syntactical pattern matching for program expressions and program statements. The graphically defined properties are then serialized into the Remopla language of Moped model-checker. Section 6.3 discusses the details of the security property specification phase.

### **Phase2: Static Analysis for Pre-processing**

- Input: Program GIMPLE representation and security properties.
- Output: Call-graph and alias information.

Given a program and set of security properties to verify, this pre-processing phase conducts call-graph analysis and alias analysis of the program. By considering the required properties, this phase identifies property-relevant behaviours of the analyzed program and discard those that are irrelevant. Besides, we resort to alias analysis in order to limit the number of tracked variables. We only consider variables that are explicitly used in security-relevant operations together with their aliases. All other variables are discarded from the verification process. More details on the static analysis pre-processing phase are given in Section 6.6. The static pre-processing phase helps generating concise models that reduce the size of state spaces to explore.

### **Phase3: Program Model Extraction**

- Input: Program source code and specified security properties.
- Output: Control-flow driven Remopla model or data-driven Remopla model.

Both the program and the specified properties are translated into Remopla representation and then combined together. The combination of program models and security properties serves the purpose of synchronizing the program behaviours with the security automaton transitions. In other words, transitions in security automata are triggered when they match the current program statement. Our verification approach carries out program model extraction in two different modes: the control-flow driven mode and the data-driven mode. The control-flow mode preserves in Remopla models the flow of control of programs, but discards data dependencies between program expressions. The resulting Remopla model is efficiently used to detect temporal security rule violations and scales to large programs. On the other hand, our data-driven model captures flow-sensitive data dependencies between program expressions. Hence, it enhances the precision of our analysis and reduces the number of false positives. Our approach for integrating data dependencies within program Remopla models is defined in Section 6.5.

#### Phase4: Property Model-Checking

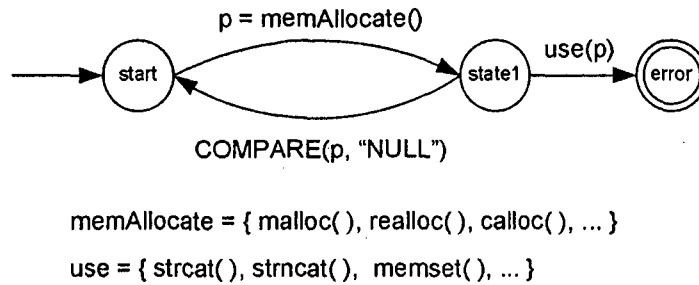
- Input: Remopla model.
- Output: Detected error traces.

The model-checking is the ultimate step of our process. The generated Remopla model is given as input to the Moped model-checker for security verification. An error is reported when a security automaton specified in the model reaches a risky state. The original version of Moped has a shortcoming in a sense that it stops processing at

---

**Figure 4** Null-checking of memory allocation functions.

---



the first encountered error. We have done modification to Moped in order to be able to detect more than one error in a run. Moreover, we have developed an error trace generation functionality that maps error traces derived from the Remopla model to actual traces derived from the source code.

## 6.3 Modeling Security Properties

In this section, we describe the modeling of temporal security properties in our framework. We detail the steps from the specification of an automaton-based property to its serialization into Remopla representation.

### 6.3.1 Temporal Security Properties

We target temporal security properties that dictate the execution order of security-relevant operations. We express such properties as finite state automata where nodes represent program states, and labeled transitions represent security-relevant program operations. A sequence of operations that reaches the final state stands for a program

**Table 19** Expression and statement pattern matching

	Transition Label Pattern	Matches
User-defined Label	f()	Invocation of functions matching pattern f().
	f("foo")	Invocation of functions in pattern f with argument constant value equal to foo.
	f(!"foo")	Invocation of functions in pattern f with argument constant value different from foo.
	x = f()	Assignment where the rhs is a function call.
	x = y	Assignment where the rhs is not a function call.
	f(*, x <sub>i</sub> , *) <sub>i=1..n</sub>	Invocation of functions in pattern f() where the <i>i</i> <sup>th</sup> parameter is security-relevant. (* indicates a sequence of irrelevant variables)
Builtin Label	ACTION_PROGRAM_START	Entry point of programs.
	ACTION_PROGRAM_END	Exit of programs.
	ACTION_FUNCTION_RETURN_f	Return from call to functions in pattern f.
	COMPARE(x, y)	Comparison between pattern X and pattern Y.

execution path that violates the security property of the automaton. As such, final states are risky states that should never be reached. For instance, the automaton of Figure 4 states that newly allocated pointers should be checked against NULL before being used. Using a pointer without performing a NULL check leads to the automaton error state.

### 6.3.2 Pattern-based Security Automata

To facilitate the specification of a wide range of properties, the transition labels of security automata support syntactical pattern matching for program variables and program statements. Table 19 summarizes the usage of patterns for automaton transition labels. There are two kinds of patterns:



- User-defined patterns for which the matching is explicitly defined by the user.

For instance, consider the label `p = memAllocate()` in the property automaton of Figure 4, the variable pattern `p` matches any pointer variable, and the operation pattern `memAllocate()` matches `malloc` functions such as `malloc()`, `realloc()`, and `calloc()`. In a given automaton, a variable pattern that appears in several transition labels must match the same variable at each occurrence. In Figure 4, the value of pattern `p` in label `p = memAllocate()` is the same in label `COMPARE(p, "NULL")`. Transition labels also support syntactic matching of constants (i.e., strings and numbers) which are specified within double quotation marks as defined in Table 19.

- Built-in patterns that are implicitly defined in our framework. The `COMPARE()` built-in pattern tracks the presence of a specific check in the program such as pointer null check or pointer bound check. The pattern takes as arguments the two operands of a comparison expression. Notice that we do not consider the boolean result of these checks. We only verify their required presence in the source code. We also have built-in patterns that match program entry point `ACTION_PROGRAM_START`, program exit `ACTION_PROGRAM_END`, and function return statements `ACTION_FUNCTION_RETURN_f`.

### 6.3.3 From Security Automata to Remopla

The second step of property specification translates a specified security automaton into a Remopla representation which we also refer to as a Remopla automaton. We

benefit from the expressiveness of the Remopla language to specify security automata as Remopla modules. In fact, these modules implement the concepts of functions and procedures in Remopla. The automaton module of Listing 6.2 illustrates the Remopla translation of the security automaton in Figure 4.

Listing 6.2: Remopla representation of the automaton in Figure 4

```

enum states {start, state1, error};
enum actions {ACTION_FUNCTION_CALL_memAllocate, ACTION_COMPARISON,
              ACTION_FUNCTION_CALL_use};
int current_state;
int p;
INITIALIZATION: current_state = start;

move_state (int action)
{
  if
  :: current_state == start ->
    if
      :: action == ACTION_FUNCTION_CALL_memAllocate
        -> current_state = state1;
        p = ARG[RETURN_VALUE_INDEX];
      :: else -> break;
    fi;
  :: current_state == state1 ->
    if
      :: action == ACTION_COMPARISON
        && ARG[0] == p && ARG[1] == NULL -> current_state = start;
      :: action == ACTION_FUNCTION_CALL_use
        && ARG[0] == p -> current_state = error;
      :: else -> break;
    fi;
  :: current_state == error -> break;
  :: else -> break;
  fi;
}

```

The nodes and the transition labels of a security automaton are mapped to Remopla constructs, as defined hereafter:

- **Remopla Automaton Nodes:** The automaton nodes are defined as elements of a Remopla enumeration variable referred to by `states`. The first line of the Remopla module of Listing 6.2 defines the states of the null check automaton. Each enumerated state corresponds to a unique integer value. We define the state values `start` and `error` to represent the automaton initial state and final

---

**Table 20** Remopla representation of program actions.

---

Automaton Alphabet	Remopla Representation
ACTION_PROGRAM_START	ACTION_PROGRAM_START
ACTION_PROGRAM_END	ACTION_PROGRAM_END
f()	ACTION_FUNCTION_CALL_f
f(V0, ..., Vn)	ARG[0]=V0; ... ; ARG[n]=Vn; ACTION_FUNCTION_CALL_f
ACTION_FUNCTION_RETURN_f	ACTION_FUNCTION_RETURN_f
COMPARE(V0, V1)	ARG[0]=V0; ARG[1]=V1; ACTION_COMPARISON
V0 = V1	ARG[0]=V0; ARG[1]=V1; ACTION_VAR_MODIFICATION

---

state, respectively. Since the security automaton actually defines the negation of a security property, the final state is the risky state. The integer variable `current_state` is used to track the current state of the security automaton. The keyword `INITIALIZATION` sets the variable `current_state` to the automaton initial state which is by default the value `start`.

- **Remopla Automaton Transitions:** The automaton transition labels are defined as elements of a Remopla enumeration variable referred to by `actions`. The second line of the Remopla module of Listing 6.2 defines the transitions of the null check automaton. Each enumerated automaton label corresponds to a unique Remopla integer constant. Table 20 shows the Remopla constructs corresponding to the automaton transition labels defined in Table 19. In fact, the mapping to Remopla constructs for program entries, program exit, function

calls without arguments, and function returns is straightforward. For program actions that involve program variables, we define a global Remopla array `ARG []` that stores the program variables in question and is inquired during the model-checking process. The two operands of assignment operations and comparison operations are put into the global array `ARG []`. The left-hand-side operand is placed at index 0 of the array, and the right-hand-side operand is placed at index 1. The parameters of function calls are placed in the array `ARG []` with respect to their position in the function argument list.

### 6.3.4 Execution of Remopla Automata

Now that we have defined the Remopla constructs for nodes and transition labels, we are able to define a Remopla module for each security automaton. In our framework, the Remopla module `move_state()` defines a property automaton. It takes as argument the current program action, and has a body that consists of a sequence of statements implementing the security automaton behaviour. The following paragraph illustrates the execution of the `move_state()` module in Listing 6.2 corresponding to the null-check property:

- **Initialization:** The variable `current_state` is set to start in order to initialize the execution of the Remopla `move_state()` module.
- **Action matching and state transition:** The `move_state()` module takes as argument the current program action that is represented by a unique integer

value as defined in Section 6.3.3. Then, given the value of `current_state`, the `move_state()` module checks if the action argument matches a transition label. Considering Listing 6.3.3, let the `current_state` be set to `start`, if the current action matches the statement `q = malloc(sz)`; then pattern `p` is set to the return pointer `q` stored in `ARG[RETURN_VALUE_INDEX]`. Besides, the `current_state` is set to `state1`.

- **Error Detection:** The `move_state()` module detects an error when the executed transition assigns the value `error` to `current_state`. Let `current_state` be equal to `state1`, when reaching the following statement `memset(q, '\0', sz)`; the action matches `ACTION_FUNCTION_CALL_use`. In addition, the pointer parameter `q` matches the value of pattern `p`. Hence, the transition to state `error` is triggered and a property violation is detected.

## 6.4 Program Model Extraction

The model extraction is the process that translates program source code to Remopla representation. First, the GCC compiler serializes the source code into its GIMPLE intermediate representation. Then, we map the GIMPLE representation to a Remopla model of the considered program. The GIMPLE representation preserves substantial information of the source code, so a simplistic conversion of all available information into program models would lead to the state space explosion problem of model-checking [38]. Hence, we utilize an abstraction technique that we define

**Table 21** Remopla representation of program.

Program Constructs	Remopla Representation
<code>any_type v;</code>	<code>int v;</code>
<code>f(){ ... }</code>	<code>module void f(){ move_state(ACTION_FUNCTION_CALL_f); ... move_state(ACTION_FUNCTION_RETURN_f); }</code>
<code>f(v<sub>0</sub>, ..., v<sub>n</sub>);</code>	<code>ARG[0]=v<sub>0</sub>;...;ARG[n]=v<sub>n</sub>; f();</code>
<code>return v;</code>	<code>ARG[RETURN_VALUE_INDEX] = v</code>
<code>v<sub>1</sub>=v<sub>2</sub>;</code>	<code>ARG[0]=v<sub>1</sub>; ARG[1]=v<sub>2</sub>; assignment();</code>
<code>if(v<sub>1</sub> op v<sub>2</sub>){ ... }else{ ... }</code>	<code>ARG[0]=v<sub>1</sub>;ARG[1]=v<sub>2</sub>; move_state(ACTION_COMPARISON); if :: true -&gt; ...; :: true -&gt; ...; fi;</code>

in this section to reduce the size of Remopla models. The program facts we take into account to construct program models are the following: variable declarations, function definitions, function calls, function returns, and control-flow structures. The handling of the aforementioned program facts is explained hereafter.

### 6.4.1 Variable Declarations and Function Definitions

Due to the limited data types provided by Remopla, we use Remopla integer type to represent all declared variables in source code. Each function definition is represented as a Remopla module with void return type and without formal parameters. The

translation of function statements follows the mapping defined in Table 21. Remopla modules with empty body are created for library functions whose source code are unavailable. A function module captures two important program actions: pattern `ACTION_FUNCTION_CALL_f` indicates the entry point of function `f()` and pattern `ACTION_FUNCTION_RETURN_f` indicates the return from function `f()`. These two actions are expressed explicitly in the function module since they may be involved in a security property and trigger a change to its corresponding automaton state. Upon each function entry and return, the property automaton module `move_state()` is invoked to check these actions against the automaton transitions and change the program state accordingly.

#### 6.4.2 Function Calls and Returns

We handle interprocedural parameter passing and return through the definition of a global Remopla array called `ARG[]`. A function call and a function return initialize the `ARG[]` array as shown in Table 21. At each function call, the `ARG[]` array stores the function parameters according to their position in the function formal argument list. The first parameter is at index 0, the second at index 1, and so forth. A function return statement stores the return value in the last entry of the `ARG[]` array indexed by `RETURN_VALUE_INDEX`. By using this global array for parameter passing, we provide the flexibility to preserve in the program model only security relevant parameters. For example, if the second parameter of a function call `f(x,y)` is of our interest, instead of translating it as `ARG[0] = x; ARG[1] = y; f();` we consider only

the second parameter and translate the call into `ARG[1] = y; f();`. As such, the number of function parameters to track is reduced to the minimum.

### 6.4.3 Flow Constructs

Control-flow skeleton should be preserved in program models for temporal property verification. At GIMPLE level, complex control structures such as `for`, `do/while`, and `switch`) are flattened and represented using `if/else` and `goto` statements. Compound conditions are also split and represented with multiple `if` blocks. This simplified GIMPLE representation eases our translation to Remopla. As shown in Table 21, an `if/else` GIMPLE construct has a corresponding Remopla `if` construct that preserves its structure. The two operands involved in the condition are stored in the `ARG[]`. As explained earlier in Section 6.3, we do not evaluate boolean values of conditions, we are only concerned by their occurrence in source code. Notice that in Remopla translation a guard condition set to `true` is attached to each branch. The model-checker Moped would explore exhaustively all conditional branches. This implies that the control flow skeleton in the model is path-insensitive in the sense that we consider all paths in the source code without pruning infeasible paths. The reason we sacrifice the analysis precision is to ensure the scalability of our framework.



#### 6.4.4 Assignments

As presented in Section 6.2.3, our framework can perform in two modes: a control-flow mode that discards all data facts, and a data-driven mode that establishes dependencies between security-relevant variables. In the data-driven mode, assignment operations are of great interest to deduce data dependencies. We define the built-in Remopla module `assignment()` to capture assignment operations and to compute dependencies between variables as detailed later in Section 6.5. Assignment statements are translated into function calls to the module `assignment()`. The two operands of an assignment are treated as call parameters and passed to the module `assignment()` using the global Remopla array `ARG[]`.

### 6.5 Dealing with Data Dependencies

Relationships between variables are important to detect violations of data-driven properties, so we need to preserve variable dependencies in Remopla models. Note that we are interested in whether variables are related, but not the actual values they are carrying. To this end, we introduce an additional Remopla integer array referred to by `stack[]`, to represent data dependencies. Given the `stack[]` array, dependencies between program variables are tracked by simulating the effects caused by the following program operations: variable declarations, assignment operations, function calls, and function returns. The following paragraphs describe our approach of modeling data dependencies using the `stack[]` array.

### 6.5.1 Variable Declarations

Each program variable is assigned an entry in the `stack[]` array. Variable identifiers serve as indexes for the array. The new entries are appended at the top of the stack array. We define the integer variable `stack pointer` that refers to the first empty array entry at the top of the stack. We also define the module `assign_entry()` that places a new entry in the stack array and increments the stack pointer by one. Initially, each added entry contains the value of its index since no data dependencies are captured yet. In fact, we model an aliasing relation between variables by assigning the same values to their entries in the `stack[]` array.

We distinguish between global variables and local variables. The former are permanently kept in the `stack[]` array, whereas the latter are stored in array `stack[]` at function calls and removed from it at function returns. More details are given hereafter:

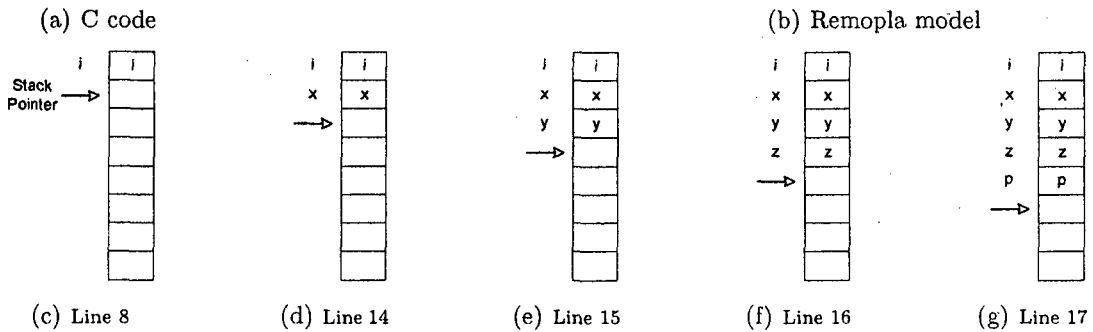
- Global variables are gathered during the parsing of the GIMPLE representation of programs. In the Remopla program model, each global variable is declared in the global scope of the model. As shown in Figure 5, the global variable `i` in source code of Figure 5(a) is defined as a global variable in Remopla model of Figure 5(b). The module `init_global_var()` assigns an entry in the `stack[]` array to each global variable of the model. This module is called during the initialization of the verification process marked by the keyword `INITIALIZATIONS` as shown in Line 2 of the Remopla module of Figure 5(b). Figure 5(c) depicts

**Figure 5** Global and local variable initializations.

```

1  int i;
2  INITIALIZATIONS:
3  init_global_var();
4  goto main;
5
6  module void init_global_var(){
7      i = assign_entry();
8  }
9
10 module void main(){
11     int x, y, z, p;
12     int local_var_number;
13     x = assign_entry();
14     y = assign_entry();
15     z = assign_entry();
16     p = assign_entry();
17     local_var_number = 4;
18     ...
19 }

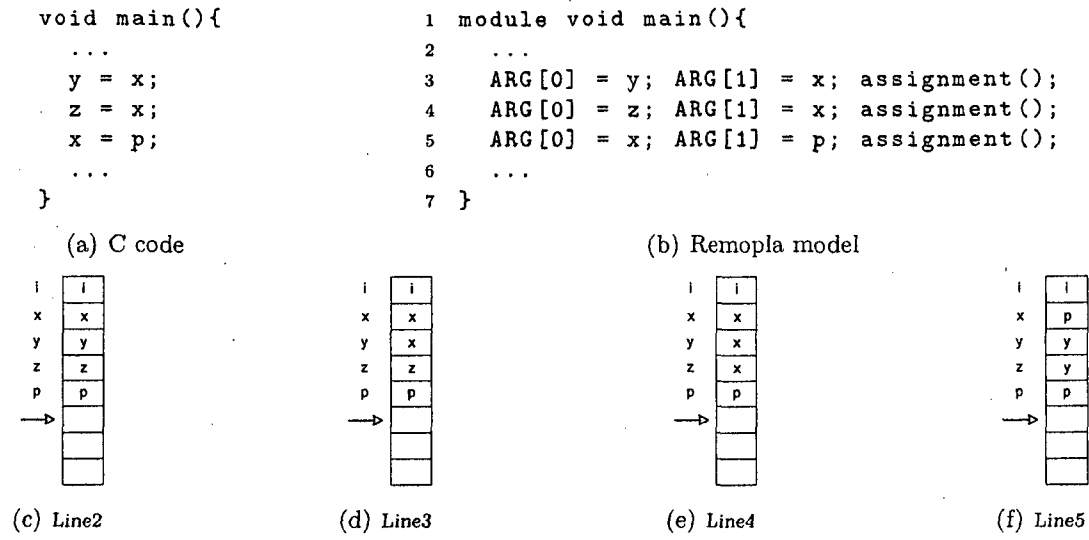
```



the `stack[]` array after the declaration of the global variable `i`. This entry remains permanently in the array during the verification process.

- Variables of local scope are assigned indexes of the array when entering a function. For instance, Figures 5(d) to 5(g) illustrate the stack array after the declaration of variables `x`, `y`, `z`, and `p`, respectively. The number of increments that are performed to the stack pointer within a function is stored in variable `local_var_number`. The latter is used to decrement the stack pointer and

**Figure 6** Assignment operations.



removes local variable entries at function returns as detailed later in this section. Notice that the stack management utilizes a scope-based approach. In other words, our analysis does not consider anymore local variables that become out-of-scope. This allows us to define a scalable approach to deal with data dependencies.

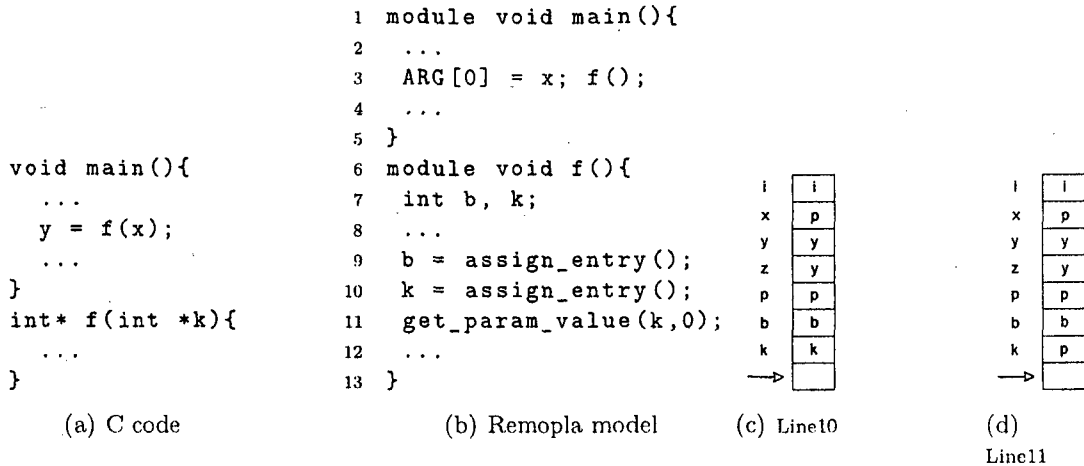
### 6.5.2 Assignment Operations

We define the built-in module `assignment()` to derive data dependencies between variables from assignment operations. Each assignment modifies the `stack[]` array to create new aliasing relations, and eventually to kill previous aliasing relations. As such, our analysis uses the array to account for aliasing information. In fact, the module `assignment()` considers the operation  $v1 = v2$  as what follows:

- The array entry indexed by `v1` is assigned the value indexed by `v2` to indicate that `v1` and `v1` are aliased. For example, in Figures 6(d) and 6(e), the entries of `y` and `z` are set to the value `x`. As such, the content of the `stack[]` array indicates that variables `x`, `y`, and `z` are aliased.
- The aliasing relations that are killed by the assignment are removed from the `stack[]` array. Among all the variables that previously referred to `v1`, we select the one with the least index value and set the value of its entry to the value of its own index. All entries of other aliases of `v1` are set to the index value of the selected variable. For example, in Figure 6(f), the entry of `x` is changed to `p` to indicate the new aliasing relation with variable `p` created at line 5 of the Remopla model. To kill the previous aliasing relation between `x`, `y`, and `z`, we change the entry of `y` to its index value. Since `y` and `z` are still aliased, the entry of `z` is set to `y` as well.

! Notice that we only consider simple assignment operations with no dereference operator. This is due to the absence of pointer type in Moped. In fact, we represent all pointers as integer variables. It is possible to extend the `stack[]` array with an additional entry containing points-to information, however this will drastically affect the performance and the scalability of Moped. We choose to sacrifice precision for the sake of scalability.

**Figure 7** Function call with parameters.

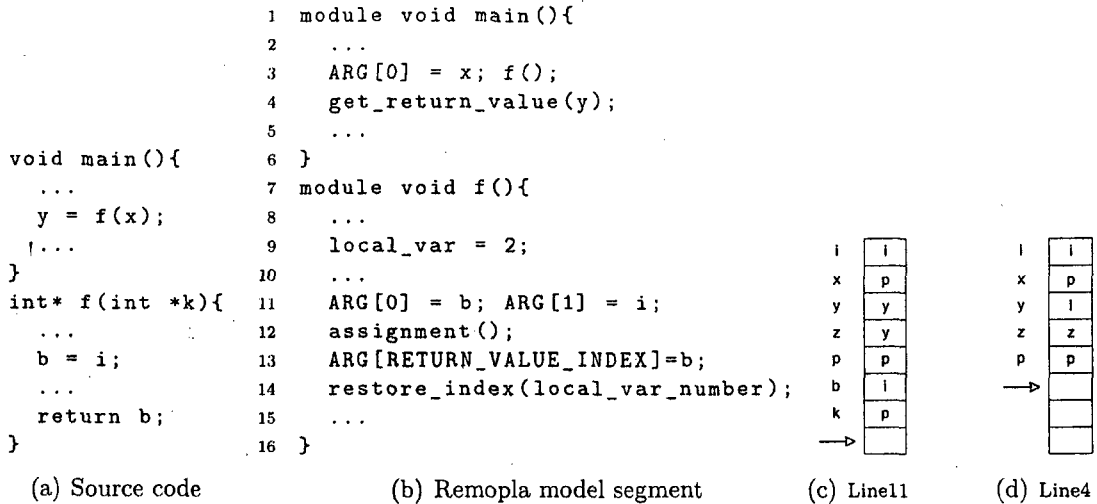


### 6.5.3 Function Call with Parameters

In what follows, we detail our approach for handling parameter passing during function calls:

- **Remopla actual parameters:** as defined in Section 6.4.2, the arguments of function calls are stored in the global array ARG[]. For instance, in Line 3 of Remopla representation of Figure 7(b), argument x of the call to function f() is put at index 0 of ARG[].
- **Remopla formal parameters:** each formal parameter in a callee function is declared as a local variable which is assigned the actual argument of the call. In Figure 7(b), the argument k of function f(int \*k) is declared as a local variable of f() in its Remopla representation. As for all local variables, k is assigned an entry in the stack[] array as depicted in Figure 7(c).
- **Matching formal parameters and actual parameters:** We define the built-in

**Figure 8 Function return.**



module `get_param_value()` to perform parameter matching at function calls. The first argument of `get_param_value()` refers to the callee function formal parameter. The second argument indicates the parameter position in the function parameter list. The call `get_param_value(k,0)` in Figure 7(b) matches the formal parameter `k` to the actual parameter at position 0 in the global array `ARG[]`. Since `ARG[0] = x`, formal parameter `k` is matched with the actual parameter `x`. The stack array of Figure 7(d) illustrates that the formal parameter `x` is aliased with the actual parameter `k`.

### 6.5.4 Function Return

The return value of a function is also passed using the `ARG[]` global array. In the Remopla model of the callee function, the return value is stored in the entry of array

ARG[] indexed by RETURN\_VALUE\_INDEX. On the caller side, the built-in module `get_return_value()` accesses the entry `ARG[RETURN_VALUE_INDEX]` to retrieve the callee return value. In Figure 8(a), the source code of `main` sets variable `y` to the return value of `f()`. In the Remopla translation of Figure 8(b), the module of `f()` sets `ARG[RETURN_VALUE_INDEX]` to variable `b`. The Remopla module of `main()` invokes `get_return_value(y)` to retrieve the value in `ARG[RETURN_VALUE_INDEX]` and to assign it to `y`. For the sake of performance, when a function returns, we remove all its corresponding local variables from the `stack[]` array. Given the number `local_var_number`, the module `restore_index()` decrements the `stack` pointer and reduces the number of stored entries. Figures 8(c) and 8(d) illustrate the stack array when entering module `f()` and when returning from module `f()`, respectively.

## 6.6 Static Analysis to Improve Abstractions

The model construction process described in the previous section produces Remopla models with large number of program functions and variables. This may lead to the state explosion problem of model-checking and render our approach unscalable to large software. To improve the performance of the verification process, the program model construction needs to be optimized. We achieve this goal by incorporating a static analysis component into our framework. By taking into account the specified properties, this component preprocesses the GIMPLE representation of source code to identify property-relevant program actions and variables that need to be preserved



---

**Table 22** Model optimization by pruning security irrelevant functions.

---

Package	LOC	Functions (CFG mode)		
		Total	Relevant	Reduced
openSSH-5.0p1	59K	1645	968	41.2%
shadow-4.1.2.2	22.7K	852	103	87.9%
apache-1.3.41	76K	1559	163	89.5%
freeradius-server-2.13	77K	2135	397	81.4%
kstart-3.14	1K	155	27	82.6%

---

in program models, excluding property-irrelevant information. The static analysis in our framework involves a call-graph analysis and an alias analysis. This section details how the static analysis pre-processing phase works in our framework.

### 6.6.1 Call-Graph Analysis

A call-graph is a directed graph that captures the interaction between functions. Each graph node represents a function, and a directed edge connecting a callee node to a caller node represents one or more invocations of the callee. Given a program call-graph, we are able to reduce the size of its corresponding Remopla model by performing the following steps:

- We extract from the call-graph the chains of successive calls in which security-relevant operations are used. All functions that are present in these extracted chains are translated into Rempola modules.
- Functions that are not invoked in security-relevant chains are not considered during the extraction of Rempola models.

Table 22 illustrates the effect of applying the call-graph analysis to all programs in the analyzed packages with respect to the null-check property of Figure 4. We extract call chains starting from the different entry points of the considered packages, and identify the security relevant ones. Table 22 shows that the number of functions that are considered is reduced by a significant amount ranging from 42% to almost 90%. Notice that the size of the generated Remopla model increases with the number of security relevant program actions listed in the transition labels of property automata. A large property automata may result in a large Remopla model provided that these actions are actually present in the source code. The complexity of the Remopla model, in terms of the number of modules it includes, is linear to the number of program functions from the considered code that contains the security relevant program actions.

### 6.6.2 Alias Analysis

In our verification framework, we resort to pointer analysis in order to reduce the number of security relevant variables that need to be tracked by the model-checker.

The pointer analysis allows us to extract the following information:

- Variables that are explicitly used in security-relevant operations and that obviously need to be considered during the construction of program models.
- Aliases of these aforementioned variables that are indirectly related to security operations. These aliases are also considered in program Remopla models.

- Variables that do not belong to the aforementioned set of variables are not considered in program models.

The literature contains several pointer analysis algorithms that can be classified into flow-sensitive alias analysis and flow-insensitive alias analysis. The former considers the execution order of statements in the program oppositely to the latter one. Flow-sensitivity increases the analysis precision at the cost of more complexity and less scalability, whereas flow-insensitivity scales to large program but with loss of accuracy and precision. Choosing either of these two types of analysis depends on the usage purposes of their output. We advocate the usage the flow-insensitive pointer analysis presented in [45] for the following reasons:

- Our verification approach is flow-sensitive and simulates execution traces of the analyzed program. Moreover, it has the capability of extracting flow-sensitive variable dependencies as detailed in Section 6.5. Thus, a flow-sensitive pre-processing alias analysis is costly and produces information that can be deduced by the model-checker itself.
- The alias analysis algorithm in [45] has a proven scalability feature [5].

From the results given in Table 23, we show that the pre-processing alias analysis helps reducing the number of tracked variables significantly. The average of the reduction is around 96%, hence the data dependencies that are derived during model-checking are related to a limited number of variables. Notice that our data

---

**Table 23** Model optimization by pruning security irrelevant variables.

---

Package	LOC	Variables (Data mode)		
		Total	Relevant	Reduced
OpenSSH-5.0p1	59K	57413	7422	87.1%
shadow-4.1.2.2	22.7K	24943	275	98.8%
apache-1.3.41	76K	50179	263	99.4%
freeradius-server-2.13	77K	76811	823	98.9%
kstart-3.14	1K	2867	111	96.1%

---

dependency algorithm does not deal with some program facts such as pointer dereference and field dereference since these operations require a more sophisticated data dependency construct to be integrated within the Remopla model of a given program. In other words, we need at least to add a new entry to the `stack []` structure to capture the level of indirection of pointers (number of dereference operators). By increasing the size of the `stack []` structure, we drastically affect the scalability of the Moped model-checker. We choose to keep the data dependency to simple but widely used constructs to achieve a better trade-off between precision and scalability of the verification process.

## 6.7 Conclusion

In this chapter, we presented our software security verification framework that combines static analysis and model-checking. The combination consists in utilizing static analysis to automatically build model-checkable abstractions of programs. It also

takes advantage of the model-checking flexibility in verifying a wide range of system-specific security properties. Our implementation is based on the TREE-SSA framework of the GCC compiler and the Moped model-checker for pushdown systems. Our tool performs security verification in two modes: (1) a control-flow mode that discards data dependencies, and (2) a data-driven mode that captures and computes data dependencies between program expressions. In the following chapter, we present the conducted experiments that demonstrate the scalability and the efficiency of our tool in detecting real errors in large C software.

# Chapter 7

## Design, Implementation, and Experimental Results

### 7.1 Introduction

This chapter demonstrates the capability of our security verification framework in detecting real errors in large scale C software packages. We show that our tool can be efficiently used for uncovering undesirable vulnerabilities in source code. The CERT secure coding website [2] is a valuable source of information to learn the best practices of C, C++, and Java programming. It defines a standard that encompasses a set of rules and recommendations for building secure code. Rules must be followed to prevent security flaws that may be exploitable, whereas recommendations are guidelines that help improve software security. The CERT standard also makes another difference between rules and recommendations stating that compliance of a code to rules

can be verified, whereas the compliance to recommendations is not always verifiable. During our experimentation, we utilize our security verification framework to check the compliance of software packages with the CERT secure coding rules. Notice that we target CERT rules that can be formally specified as finite-state automata and given as input to our framework. These automata-based rules represent a wide majority of the CERT standard. The experiments presented in this chapter are conducted in the two modes of our security verification tool: the control-flow mode that discards data dependencies and the data-driven that establishes data dependencies between program variables. The hardware platform used for the experiments is a Dell D810 with Pentium M 1.86GHz CPU and 1G memory that runs Fedora Core 8.

This chapter is organized as follows: We give an overview of our Section 7.2 gives an overview of our tool implementation and the CERT coding rules used in our experiments. Our conducted experiments in the control-flow mode are detailed in Section 7.3. The results of our experiments in the data-driven mode are presented in Section 7.4. We draw conclusion in Section 7.6.

## 7.2 Design and Implementation

In this section, we motivate our choice of using GIMPLE representation of source code and the conventional pushdown model-checker Moped. We also give an overview of the CERT secure coding rules used to conduct our experiments.

### 7.2.1 Why GIMPLE and Moped ?

Our ultimate goal is to provide a security verification tool for open source software, thus we base our approach on the GCC compiler considered as the defacto open source compiler. The GCC mainline recently includes the Tree-SSA framework [97] that facilitates static analysis with its GIMPLE intermediate representation of source code. In fact, GIMPLE linearizes all high-level control flow structures including nested functions, exception handling, and loops. Working with GIMPLE representation allows our analysis to be focused more on data modifications and control-flow information instead of putting effort into analyzing complex language constructs. Besides, the language- and platform-independent features of GIMPLE provide appealing flexibility features for our approach to be extended to all GCC supported languages.

For the verification process, we use the Moped model-checker for pushdown systems that are known to efficiently model programs' execution stack and interprocedural behaviours [50]. Moped has a C-like input language called Remopla to define programs as pushdown systems. The procedural nature of Remopla facilitates the translation of a GIMPLE representation to a Remopla model. As such, we benefit from static analysis to automatically build model-checkable abstractions of large scale programs. Automata-based security properties and program Remopla model are input to our security verification tool in order to detect security violations and provide witness paths leading to them.



## 7.2.2 Macro Handling

The GIMPLE representation of programs is closely related to the environment under which the program is compiled. This tight coupling between the underneath environment and the considered code gives an appealing precision feature to our analysis compared to other approaches directly based on source code. Consider the following code snippet in Listing 7.1 taken from the `binutils-2.19.1` package. For code portability purposes, the macro `HAVE_MKSTEMP` is checked in `#ifdef` to verify whether the system supports function `mkstemp()` for safe temporary file creation. If not, function `mktemp()` is used instead. A simplistic traversal of the source code would flag an error for the occurrence of `mktemp()` considered as an unsafe function for temporary file creation. Being based on GIMPLE representation, our analysis does not suffer this false alert. In fact, GIMPLE representation solves the conditional `#ifdef`, and one of the two temporary file functions will appear in the GIMPLE code with regard to the compilation environment. In our case, the machine used to conduct the experiments supports `mkstemp()` which is present in the GIMPLE code of Listing 7.2

Listing 7.1: Sample C code from `binutils-2.19.1` with macros

```
#ifdef HAVE_MKSTEMP
    fd = mkstemp (tmpname);
#else
    tmpname = mktemp (tmpname);
    if (tmpname == NULL)
        return NULL;
    fd = open (tmpname, O_RDWR | O_CREAT | O_EXCL, 0600);
#endif
```

Listing 7.2: GIMPLE representation of code in Listing 7.1

```
D.8401 = mkstemp (tmpname);
fd = D.8401;
if (fd == -1)
    { D.8402 = 0B;
      return D.8402;
    }
```

This points out the important fact that the verification of software should be performed on the same environment intended for their real usage. Besides, the verification should be performed on hostile environments to predict as much worst execution scenarios as possible.

### 7.2.3 Temporary Variables

The GIMPLE representation breaks down program expressions into SSA form in which each variable is defined exactly once [97]. This form of representation involves the definition of temporary variables that hold intermediate values. Consider the call to `malloc()` function in Listing 7.3, its corresponding GIMPLE code in Listing 7.4 splits the `malloc()` call into two sub-expressions involving a temporary variable `D.1861`.

Listing 7.3: Sample C with memory allocation

```
p = malloc(BUFSIZ)
if (!p)
    return -1;
...
free(p);
return 1;
```

Listing 7.4: GIMPLE representation of code in Listing 7.3

```
D.1860 = malloc(5);
p = (char *) D.1860;
if ( p == 0B) {
    D.1861 = -1;
    return D.1861;
}
...
free(p);
D.1861 = 1;
return D.1861;
```

The return value of `malloc()` is assigned to a temporary variable `D.1861`. Then, the latter is cast and assigned to pointer `p`. The usage of temporary variables presents

a challenge for pattern matching. In this example, variable `D.1861` matches the pattern for the return value of `malloc()`, whereas variable `p` matches the pattern for the call to `free()` argument. Without considering relations between temporary variables, the verification process flags an erroneous warning for the deallocation of an uninitialized pointer. The expressiveness of the GIMPLE representation helped us to overcome this challenge. In fact, GIMPLE keeps track of the original definition of temporary variables. In the given example, we are able to recognize that temporary variable `D.1861` is an intermediate representation of `p` and avoid spurious warnings.

#### 7.2.4 CERT Coding Rules

To assist programmers in the verification of their code, we integrate in our tool a set of secure coding rules defined in the CERT standard [2]. The objective is to provide programmers with a framework to evaluate the security of their code without the need to have high security expertise. The CERT secure coding rules can be mainly classified into the following categories:

- *Deprecation rules*: These rules are related to the deprecation of legacy functions that are inherently vulnerable such as `gets()` for user input, `tmpnam()` for temporary file creation, and `rand()` for random value generation. The presence of these functions in the code should be flagged as a vulnerability. For instance, CERT rule MSC30-C states the following "*Do not use the `rand()` function for generating pseudorandom numbers* "

- *Temporal rules:* These rules are related to the bad sequencing of program actions in source code. For instance, the rule MEM31-C from the CERT entails to "*Free dynamically allocated memory exactly once*". Consecutive free operations on a given memory location represent a security violation. Intuitively, these kind of rules are modeled as finite state automata where state transitions correspond to program actions. An unsafe sequence of operations should lead to an error state in its corresponding automaton.
- *Type-based rules:* These rules are related to the typing information of program expressions. For instance, the rule EXP39-C from the CERT states the following "*Do not access a variable through a pointer of an incompatible type*". A type-based analysis can be used to track violations of these kind of rules.
- *Structural rules:* These rules are related to the structure of source code such as variable declarations, function inlining, and macro invocations. For instance, rule DCL32-C entails to "*Guarantee that mutually visible identifiers are unique*". As an application of this rule, the first characters in variable identifiers should be different to prevent confusion and facilitates the code maintenance.

Our approach covers the two first categories of coding rules that we can formally model as finite state automata. In fact, we cover 31 rules out of 97 rules in the CERT standard. We also cover 21 recommendations that can be verified according to CERT. Notice, that the security properties we handle are the most relevant for building secure software since these 31 rules correspond to all rules of the C language

given by the BSI (Build Security In: Homeland Security) [1]. The latter is listed as a related source of information in the CERT website [2].

## 7.3 Experiments in Control-Flow Mode

In this section, we detail our conducted experiments that consist in verifying a set of well-known and widely used open-source software against a set of CERT secure coding rules. We strive to cover different kinds of security coding errors that skilled programmers may inadvertently produce in their code. In the sequel of this section, we detail the results of the experimentation that we conducted on large scale C software. The content of the tables that present the experimentation results is described in the following paragraph. The three first columns define the package name, the size of the package, and the program that contains coding errors. The number of reported error traces is given in the fifth column (Reported Errors). After manual inspection of the reported traces, we classify them into the three following columns: column (Err) for potential errors, column (FP) for false positive alerts, and column (DN) for traces that are undecidable with manual inspection. The checking time of programs is given in the last column.

### 7.3.1 Unchecked Return Values

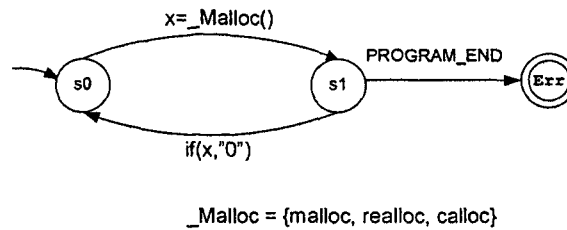
CERT Coding Rules:

- MEM32-C: Detect and handle memory allocation errors.
- EXP34-C: Ensure a null pointer is not dereferenced.

---

Figure 9 Null check automaton

---



Unfortunately, programmers very often omit to handle erroneous return values from function calls. They make wrong assumptions on the successful termination of callee functions. According to the Coverity scan report, the use of unchecked return values represents 25% of programming errors [43]. Error handling omission can lead to system crashes especially for memory allocation functions that return null pointer on failure. Therefore, rule MEM32-C entails that the return value of memory allocation functions should be checked before being used to prevent the nasty dereference of null pointers. Besides, rule EXP34-C emphasizes that null pointers should not be dereferenced. Table 24 illustrates the analysis results of the security automaton depicted in Figure 9.

We reviewed the reported error traces and mark them all as real errors. They contain a allocation operation that is never followed by a null check of the returned pointer. We give in Listing 7.5 a code snippet from the apache-1.3.41 that uses the return pointer of malloc() without null check.

Listing 7.5: Use with null-check in apache-1.3.41

```
con = malloc(concurrency * sizeof(struct connection));  
memset(con, 0, concurrency * sizeof(struct connection));
```

---

**Table 24** Return value checking.

---

Package	LOC	Program	Reported Errors	Checking time (Sec)
amanda-2.5.1p2	87K	chg-scsi	1	28.87
apache-1.3.41	75K	ab	1	0.4
bintuils-2.19.1	986K	ar	1	0.74
freeradius-2.1.3	77K	radeapclient	1	1.06
httpd-2.2.8	210K	ab	1	0.5
openca-tools-1.1.0	59K	openca-scep	2	2.6
shadow-4.1.2.2	22.7K	groupmems	1	3.08
		groups	1	2.81
		usermod	1	2.82
		id	1	2.80
		useradd	1	2.81
vipw	1	3.05		
zebra-0.95a	142K	ospf6test	1	15.13

---

### 7.3.2 Memory Leak Errors

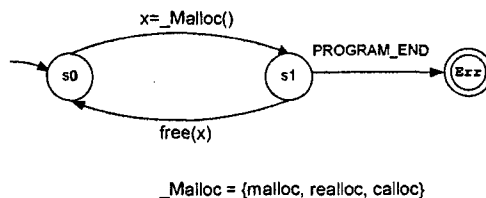
CERT Coding Rules:

- MEM31-C: Free dynamically allocated memory exactly once.

Memory locations that are assigned to processes may contain sensitive data that should not be disclosed to unauthorized users. A process must ensure that its memory locations are released and cleaned up when no longer used. Failure to do so results in resource leaks that reduce the performance and the availability of the running system. In fact, operating systems limit the size of memory space a process can own to defend against resource exhaustion. When the limit is reached, the process is not able to execute and suffers a denial of service.

To prevent memory leaks, the CERT rule MEM31-C states that all dynamically allocated memory locations should be freed before termination of a process. The automaton given in Figure 10 is used to verify the absence of memory leaks in software

Figure 10 Memory leak automaton



packages listed in Table 25. Our tools reports an error for all paths that allocate memory locations without freeing them prior to exit. We illustrate the different cases through examples in the following paragraphs.

Sample code of Listing 7.6 illustrates a real memory leak error in shadow-4.1.2.2 where pointer `buf` is allocated but never freed.

Listing 7.6: Memory leak in shadow-4.1.2.2

```
char *buf;
buf = (char *) malloc (strlen (editor) + strlen (fileedit) +2);
snprintf (buf, strlen (editor) + strlen (fileedit) + 2,
          "%s %s", editor, fileedit);
if (system (buf) != 0) {
    fprintf (stderr, "%s: %s: %s\n", progname, editor,
            strerror (errno));
    exit (1);
}
else
    exit (0);
```

Listing 7.7: False alert of memory leak in openssh-5.0p1

```
ac = ssh_get_authentication_connection();
if (ac == NULL) {
    fprintf(stderr, "Could not open a connection to your authentication agent.\n");
    exit(2);
}
AuthenticationConnection * ssh_get_authentication_connection(void){
    auth = xmalloc(sizeof(*auth));
    ...
    return auth;
}
```

The code in Listing 7.7 is extracted from the package openssh-5.0p1 for which our tool reports a false positive. In this code, function `ssh_get_authentication_connection()` returns an allocated pointer `ac`. On allocation failure, the program



**Table 25** Resource leak errors

Package	LOC	Program	Reported Errors	Err	FP	DN	Checking time (Sec)
amanda-2.5.1p2	87K	genversion	5	0	4	1	0.24
		chg-scsi	1	0	0	1	12.7
		amdd	1	1	0	0	2.83
		amidxtaped	2	0	0	2	11.24
apache-1.3.41	75K	logresolve	5	0	1	4	0.23
		httpd	2	1	0	1	2.23
		ab	5	1	0	4	0.17
at-3.1.10	2.5K	atd	4	0	4	0	0.23
		at	2	1	0	1	0.46
bintuils-2.19.1	986K	gprof	1	0	0	1	0.68
		readelf	2	0	1	1	2.01
		rablib	1	0	0	1	0.45
freeradius-2.1.3	77K	radeapclient	1	0	0	1	0.71
httpd-2.2.8	210K	ab	2	1	0	1	0.19
		logresolve	5	0	1	4	0.2
		dftables	1	1	0	0	0.14
inn-2.4.6	89K	ninpaths	2	0	0	2	0.51
openca-tools-1.1.0	59K	openca-scep	4	0	0	4	1.31
openSSH-5.0p1	58K	ssh-add	1	0	1	0	2.18
		ssh-keysign	1	1	0	0	2.55
		ssh-keyscan	2	0	0	2	2.34
shadow-4.1.2.2	22.7K	groupmems	2	0	0	2	2.33
		groups	2	1	0	1	2.2
		usermod	2	1	0	1	2.89
		useradd	3	1	0	2	2.78
		vipw	2	1	1	0	2.29

exits. Since our tool is path insensitive, it cannot determine that the function exits when pointer `ac` is null and reports a false alarm for the memory leak of pointer `ac`.

### 7.3.3 Use of Deprecated Functions

CERT Coding Rules:

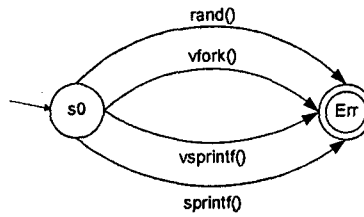
- FI033-C: Detect and handle input output errors resulting in undefined behaviour.
- POS33-C: Do not use `vfork()`.

- MSC30-C: Do not use the `rand()` function for generating pseudorandom numbers

Deprecated functions are quite abundant in the C library of functions. The CERT coding rules forbid the usage of these functions as they are readily vulnerable to attacks such as buffer overflows, code injection, and privilege escalation. The usage of safe alternatives is required as a preventive measure. We present hereafter the set of CERT rules that our tool is able to verify:

- Rule MSC30-C for random number generation: The `rand()` function produces numbers that can easily be guessed by attackers and should never be used especially for cryptographic purposes. The CERT recommends using function `random()` instead.
- Rule POS33-C for process management: The `vfork()` function suffers race conditions and denial of service vulnerabilities and should never be used. Programmers should consider the usage of `fork()` as a safe alternative.
- Rule FI033-C for string manipulation: The CERT deprecates the usage of function `gets()`, `sprintf()`, and `vsprintf()` since they are extremely vulnerable to buffer overflow attacks. Microsoft developed safe alternatives to C string functions that are documented in the technical report ISO/IEC TR 24731-1 [57]. The CERT standard STR07-C recommends the usage of these functions for the following reasons: (1) They discard the nasty "%n" format string that attackers use to overwrite memory locations with their malicious code. (2) They take as

**Figure 11** Deprecated functions automaton



argument a buffer size of type `rsize_t` that should not be larger than `RSIZE_`  
`MAX`. The latter determines the maximum memory size a single object can have.

(3) They ensure that strings are null-terminated to avoid buffer overflows.

**Table 26** Usage of deprecated functions

Package	LOC	Program	Rule	Reported Erros	Err	Checking time (Sec)
apache-1.3.41	75 K	htpasswd	MSC30-C	2	2	0.25
inetutils-1.6	276 K	rcp	POS33-C	1	1	0.47
krb5-1.6	276 K	rcp	POS33-C	1	1	0.08
		kshd	FI033-C	many	many	0.20
zebra-0.95a	142 K	ripd	MSC30-C	1	0	0.17
emacs-22.3	242 K	update-game -score	MSC30-C	1	0	0.30
wget-1.11.4	24.5 K	wget	FI033-C	many	many	0.20
chkconfig-1.3.30c	4.46 K	chkconfig	FI033-C	many	many	0.34

During our conducted experiments, we flag the occurrence of deprecated functions in the analyzed packages as an error that should be fixed. The automaton for the detection of deprecated functions is given in Figure 11. From the analysis results in Table 26, we deduce that deprecated functions are still used in many software. As illustrated in Listing 7.8, function `rand()` is used in package `apache-1.3.41` for password generation.

Listing 7.8: Unsafe usage of rand() for password generation in apache-1.3.41

```
/*
 * Make a password record from the given information. A zero return
 * indicates success; failure means that the output buffer contains an
 * error message instead.
 */
static int mkrecord(char *user, char *record, size_t rlen, char *passwd, int alg){
    char *pw;
    char cpw[120];
    char salt[9];
    if (passwd != NULL) {
        pw = passwd;
    }
    /*...*/
    switch (alg) {
        /*...*/
        case ALG_APMD5:
            (void) srand((int) time((time_t *) NULL));
            ap_to64(&salt[0], rand(), 8);
            salt[8] = '\0';

            ap_MD5Encode((const unsigned char *)pw, (const unsigned char *)salt,
                        cpw, sizeof(cpw));

            break;
        /*...*/
    }
}
```

In the case of packages zebra-0.95a and emacs-22.3, rand() is used for time synchronization purposes. Listing 7.9 shows the usage of rand() in the routing package zebra-0.95 to compute a time jitter. We do not know whether the timing for these programs are security relevant and cannot claim that the use of rand() is an exploitable error.

Listing 7.9: Using rand() to compute time jitter in zebra-0.95

```
rip_update_jitter (unsigned long time)
{
    return ((rand () % (time + 1)) - (time / 2));
}
void
rip_event (enum rip_event event, int sock)
{
    int jitter = 0;

    switch (event)
    {
        /*...*/
        jitter = rip_update_jitter (rip->update_time);
        rip->t_update =
        /*...*/
    }
}
```

### 7.3.4 Unsafe Environment Variables

CERT Coding Rules:

- STR31-C: Guarantee that storage for strings has sufficient space for character data and the null terminator.
- STR32-C: Null-terminate byte strings as required.
- ENV31-C: Do not rely on an environment pointer following an operation that may invalidate it.

String manipulation in C programming is famous for spawning exploitable errors in source code such as inappropriate format string, buffer overflows, string truncation, and not null-terminated strings. For our experiments, we focus on the following CERT rules:

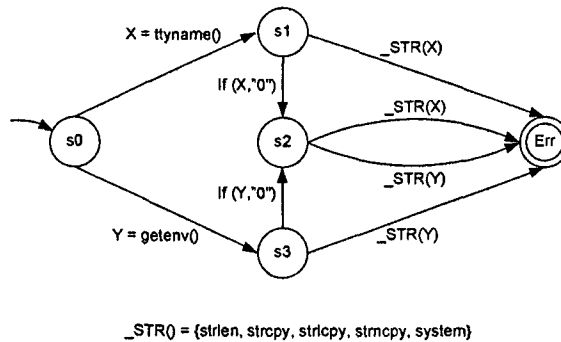
- Rule STR31-C disciplines the usage of string copy functions to prevent buffer overflows and truncation errors that arise from copying a string to a buffer that is not large enough to hold it.
- Rule STR32-C stresses on the need of a null character to mark the end of a string. For flexibility sake, the C language does not limit string sizes and depends on the presence of a null character "\0 " to mark the end of a string. The absence of this character results in buffer overflows and denial of service attacks.
- Rule ENV31-C targets the safe usage of environment functions to prevent bad assumption resulting from inconsistent environment values.

The risk of string errors increases even more when using string pointers that refer to the values of environment variables. In fact, programs' execution environment should

---

**Figure 12** Environment function automaton

---



never be trusted and should be considered as hostile to safe execution. From this conservative assumption, all values requested from the environment should be checked before usage: null pointer checks, bound checks, and null-termination checks. The C library contains a set of environment functions that are widely used despite their notorious reputation of being unsafe. Among these functions, we have `ttynam()` and `getenv()`. These functions return a string with unknown size that may be not null-terminated. On failure, these functions return a null pointer. Besides, these functions are not reentrant. In other words, if multiple instances of the same function are concurrently running, it may lead to inconsistent states. Attackers may take advantage of this reentrant characteristic to invalidate the values of environment variables. The CERT rule ENV31-C targets the safe usage of environment functions to prevent bad assumption resulting from inconsistent environment values.

We define the automaton in Figure 7.18 to detect the unsafe usage of environment functions. As explained earlier, we consider all string functions that are not part of the TR 24731-1 [57] as unsafe functions. So, using a variable that is returned from

---

**Table 27** Unsafe environment variables

---

Package	LOC	Program	Rule	Reported Errors	Err	Checking time (Sec)
openssh-5.0p1	58K	sshd	STR31-C	1	0	0.15
krb5-1.6	276K	kshd	ENV31-C	2	2	0.33
		kshd	STR32-C	2	2	0.33
patchutils-0.1.5	1.3K	interdiff	STR32-C	1	1	0.06
kstart-3.14	4.4K	krenew	STR32-C	1	1	0.06
inetutils-1.6	276 K	tftp	STR31-C	1	1	0.54
		telnet	STR31-C	1	1	0.52
chkconfig-1.3.30c	4.46 K	chkconfig	STR32-C	1	1	0.52
freeradius-2.1.3	77 K	radiusd	STR32-C	1	1	0.52

---

environment functions as an argument of an unsafe string function is considered as a vulnerability.

Table 27 illustrates the results of our experimentation for a given set of software. The fifth column indicates the reported error traces. After inspecting the traces, we distinguish false positives from what we believe to be a potential error in the sixth column. We discuss in the following paragraphs some of the reported errors.

The code in Listing 7.10 is taken from program `sshd` of `openssh-5.0p1`. It triggers a warning when analyzed with our tool. In fact, the return value `name` of `ttyname()` is copied using the function `strncpy()`. This function ensures the null-termination of the destination buffer `namebuf` provided that `namebuflen` is properly set. In other words, large enough to read all characters in `name`, though it should not overflow `namebuf`. If the size of `name` is bigger than `namebuflen`, then there is a possible string truncation error as mentioned in the programmers comments. From their comments, we assume that programmers intentionally did not handle the possible

string truncation as they do not consider it as an exploitable error. We consider this error trace as a false positive.

Listing 7.10: Unsafe usage of `ttyname()` in `openssh-5.0p1` (Rule STR31-C)

```
name = ttyname(*ttyfd);
if (!name)
    fatal("openpty returns device for which ttyname fails.");
strcpy(namebuf, name, namebuflen);    /* possible truncation */
return 1;
```

The code fragment of Listing 7.11 is taken from `krb5-1.6`. It is a good example to show what not to do when using environment variables. It calls `getenv()` to get the value of environment variable `KRB5CCNAME`.

Listing 7.11: Unsafe usage of `getenv()` in `krb5-1.6` (Rules ENV31-C and STR32-C)

```
if (getenv("KRB5CCNAME")) {
    int i;
    char *buf2 = (char *)malloc(strlen(getenv("KRB5CCNAME"))
                                +strlen("KRB5CCNAME=")+1);
    if (buf2) {
        sprintf(buf2, "KRB5CCNAME=%s",getenv("KRB5CCNAME"));
        ...
    }
}
```

In this code, `getenv()` is called three consecutive times. There is absolutely no guarantee that these three calls return the same value. An attacker may take advantage of the time race between each call to modify the value of variable `KRB5CCNAME`.

- Between the first and the second call, an attacker can remove variable `KRB5CCNAME` from the environment and the second call to `getenv()` returns a null pointer. In that case, function `strlen()` would have a null argument and would generate a segmentation fault.
- Besides, `getenv()` is used a third time as an argument to `sprintf()` which is vulnerable to buffer overflow and should be avoided according to CERT rule FI033-C. We assume that the allocation of `buf2` is successful. Between the





to redirect the access operation to another file. Figure 13 illustrates the automaton for race condition detection. It flags a check function followed by a subsequent use function as a TOCTTOU error. The analysis results are given in Table 28.

**Table 28** File race condition TOCTTOU

Package	LOC	Program	Reported Errors	Err	FP	DN	Checking time (Sec)
amanda-2.5.1p2	87K	chunker	1	0	1	0	71.6
		chg-scsi	3	2	1	0	119.99
		amflush	1	0	0	1	72.97
		amtrmidx	1	1	0	0	70.21
		taper	3	2	1	0	84.603
		amfetchdump	4	1	0	3	122.95
		driver	1	0	1	0	103.16
		sendsize	3	3	0	0	22.67
		amindexd	1	1	0	0	92.03
at-3.1.10	2.5K	atd	4	3	1	0	1.16
		at	4	3	1	0	1.12
bintuils-2.19.1	986K	ranlib	1	1	0	0	2.89
		strip-new	2	0	1	0	5.49
		readelf	1	1	0	0	0.23
freeradius-2.1.3	77K	radwho	1	1	0	0	1.29
inn-2.4.6	89K	nnrpd	1	1	0	0	4.11
		fastrm	1	1	0	0	0.37
		archive	1	0	1	0	0.95
		rnews	1	1	0	0	0.57
openSSH-5.0p1	58K	ssh-agent	2	0	0	2	22.46
		ssh	1	0	1	0	100.6
		sshd	6	3	1	2	486.02
		ssh-keygen	4	4	0	0	87.28
		scp	3	2	0	1	87.95
shadow-4.1.2.2	22.7K	usermod	3	1	0	2	9.79
		useradd	1	1	0	0	11.45
		vipw	2	2	0	0	10.32
		newusers	1	1	0	0	9.2
zebra-0.95a	142K	ripd	1	1	0	0	0.46

Listing 7.12 illustrates a race condition error in package zebra-0.95a. The `stat()` function is called on file `fullpath_sav` before being accessed by calling function

open(). Being based on pathname instead of file descriptor renders these functions vulnerable to TOCTTOU attacks as detailed in Section 2.2 of Chapter 2.

Listing 7.12: File race condition in zebra-0.95a

```
if (stat (fullpath_sav, &buf) == -1) {
    free (fullpath_sav);
    return NULL;
}
...
sav = open (fullpath_sav, O_RDONLY);
...
while(( c = read (sav, buffer, 512)) > 0)
...

```

Listing 7.13 contains a sample code that is extracted from package amanda-2.5.1p2. The mkholdingdir() function is used inside a loop. Our tool goes through the loop and considers that there is a path where stat(diskdir,...) is a check function and mkdir(diskdir,...) is a use function that corresponds to the pattern of TOCTTOU errors. We actually consider this reported error as a false positive since there are paths where the mkdir() call does not depend on the result of the stat() check. Besides, the return value of the mkdir() is used to check the successful creation of the directory.

Listing 7.13: False positive TOCTTOU in amanda-2.5.1p2

```
while (db->split_size > (off_t)0
      && dumpsize >= db->split_size)
{
    ...
    mkholdingdir(tmp_filename);
    ...
}
mkholdingdir(char * diskdir){
    struct stat stat_hdp;
    int success = 1;
    ...
    else if (mkdir(diskdir, 0770) != 0 && errno != EEXIST)
    {...}
    else if (stat(diskdir, &stat_hdp) == -1)
    { ...

```

### 7.3.6 Unsafe Temporary File Creation

CERT Coding Rule:

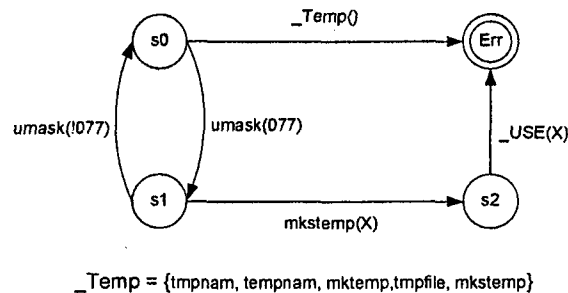
- FI043-C: Do not create temporary files in shared directories.

Very often software applications create and maintain temporary files for different purposes such as information sharing, temporary data storing, and computation speeding up. Usually applications store temporary files in shared folders, then terminate execution and leave these files behind. This bad management of temporary files exposes private and sensitive data and offers to attackers the possibility to hijack temporary files and tamper with their content. The impact of such attacks is very high especially when these targeted files are set with high privileges. Therefore, programmers must properly create, protect, and delete temporary files. The standard C library provides a set of functions for temporary file creation. However, most of these functions are vulnerable to various forms of attacks and must be used with precaution. We detail in the following paragraphs the temporary file discipline entailed by the CERT rule FI043-C and modeled in the automaton of Figure 14. Table 29 gives the verification results for a set of packages against the security rule FI043-C.

---

**Figure 14** Temporary file security automaton

---



**Table 29** Temporary file errors

Package	LOC	Program	Reported Errors	Checking time (Sec)	CERT Rule
openssh-5.0p1	58K	ssh-keygen	1	9.21	FI043-C-2
		sshd	1	50.6	FI043-C-3
		ssh-rand-helper	1	7.52	FI043-C-3
apache-1.3.41	75K	htpasswd	1	0.13	FI043-C-1
		htdigest	1	0.09	FI043-C-1
shadow-4.1.1	22.7K	useradd	1	2.75	FI043-C-2
patchutils-0.1.5	1.3K	interdiff	3	0.17	FI043-C-{1,2}
		filterdiff	1	0.11	FI043-C-1
krb5-1.6	276K	kprop	1	0.11	FI043-C-1
kstart-3.14	4.4K	k4start	1	0.14	FI043-C-2
		k5start	1	0.15	FI043-C-2
		krenew	1	0.11	FI043-C-2
chkconfig-1.3.30c	4.46K	chkconfig	1	0.11	FI043-C-1
inn-2.4.6	89K	nntpget	1	0.32	FI043-C-2
		shrinkfile	1	0.27	FI043-C-2
		innxmit	1	0.52	FI043-C-2
		makehistory	2	0.37	FI043-C-2
binutils-2.19.1	986K	ranlib	1	1.9	FI043-C-2
emacs-22.3	986K	update-game-score	1	0.19	FI043-C-3

- Temporary file creation: A temporary file must have a unique name to avoid collisions with existing files. The C functions `tmpnam()`, `tempnam()`, `tmpfile()`, and `mktemp()` generate a unique file name when invoked. However, these functions suffer a race condition between the file name generation and the file creation that can be exploited by attackers. We refer to this error as FI043-C-1 in Table 29.
- Setting appropriate permissions: Since temporary files are usually created in shared folders, it is highly required to set appropriate permissions to these files to protect them against attackers. As such, a call to `umask(077)` must be done

before a call to `mkstemp()` to limit the permissions of the resulting temporary file to only the owner. We refer to this error as FI043-C-2 in Table 29.

- Race conditions: Functions that create temporary files are considered as check functions, as defined in Section 7.3.5, that are subject to race condition errors when their filename argument is used in a subsequent system call. We refer to this error as FI043-C-3 in Table 29.

The sample code in Listing 7.14 is taken from `make-3.81` package. The GIMPLE representation of that code is given in Listing 7.15. This code is quite similar to the code fragment in Listing 7.1.

Listing 7.14: Temporary file error in `emacs-22.3`

```
#ifdef HAVE_MKSTEMP
    if (mkstemp (tempfile) < 0
#else if (mktemp (tempfile) != tempfile
#endif
    || !(f = fopen (tempfile, "w")))
return -1;
```

Both codes use the `#ifdef` macro to verify the system support of function `mkstemp()`. Otherwise, the system uses `mktemp()`. Checking for system supports of safe functions is a good practice for secure programming. However, this fragment is not error free. Suppose that `mkstemp()` is used, its file name argument should never appear in any subsequent system call according to the CERT rule FI043-C. Hence, the call to `fopen()` with the same file name presents a file race condition error detailed in Section 7.3.5.

Listing 7.15: GIMPLE representation of source code in Listing 7.14

```
D.4565 = mkstemp (tempfile);
if (D.4565 < 0) { goto <D4563>; }
/*...*/
D.4566 = fopen (tempfile, &"w"[0]);
f = D.4566;
```

In Listing 7.1, `fopen` is called only when `mktemp()` is used for the temporary file creation. The `O_EXCL` flag provides an exclusive access to the file to prevent unauthorized access. The error that we trigger for this code is related to non usage of the `umask(077)` call to set the temporary file permissions.

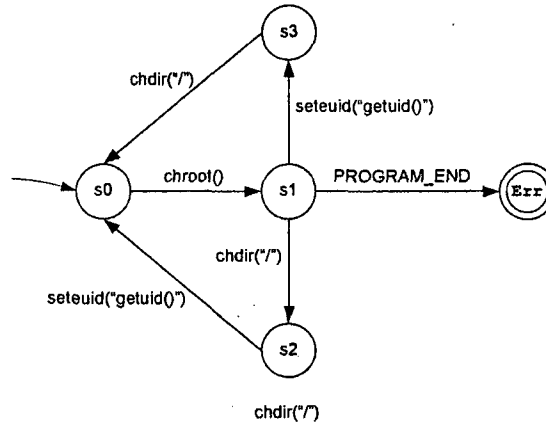
### 7.3.7 Unsafe Creation of `chroot` Jail

CERT Recommendations:

- POS02-C: Follow the principle of least privilege.
- FIO16-C: Limit access to files by creating a jail.

CERT recommendations POS02-C and FIO16-C highly advise to follow the principle of least privilege to secure processes execution. By the principle, a process should have access only to the resources required for its execution. All other accesses should be discarded. The main intent of this principle is to prevent privilege escalation threats that may occur during a potential exploit of vulnerable processes. To follow the least privilege principle, the C function provides the `chroot(new_root)` function to create a virtual root directory for the owning process. After being re-rooted, the process cannot access files outside the directory tree defined by the new root directory. Unfortunately, the programmers often commit mistakes when using the `chroot()` function as detailed in Section 2.2 of Chapter 2. The security automaton related to the considered CERT recommendations is given in Figure 15. In Table 30, we give the results of verifying a set of software against the `chroot()` related automaton.

**Figure 15** Secure creation of chroot jail automaton



**Table 30** Unsafe call to chroot ()

Package	LOC	Program	Reported Errors	Err	FP	Checking time (Sec)	CERT Rule
freeradius-2.1.3	77K	radiusd	1	0	1	0.15	FI016-C
shadow-4.1.2.2	22.7K	all	1	1	0	0.14	POS02-C

We briefly recall the common errors when invoking `chroot()` and discuss the reported errors hereafter:

- Failing to call `chdir("/")` after calling `chroot(new_root)` will prevent the process from being redirected to the confined directory. The chroot jail is ineffective and the process can still access files outside of it. In Listing 7.16, we show a false positive warning issued by our tool when verifying `freeradius-2.1.3`. As mentioned in their comments, programmers intentionally do not call `chdir("/")` to allow access to some configuration files. Though it is not a safe programming style, we consider this error as a false positive.



Listing 7.16: False positive chroot() error in freeradius-2.1.3

```
if (chroot(chroot_dir) < 0) {
    fprintf(stderr, "%s: Failed to perform chroot %s: %s",
            progname, chroot_dir, strerror(errno));
    return 0;
}
/*
 * Note that we leave chdir alone. It may be OUTSIDE of the root.
 * This allows us to read the configuration from "-d./etc/raddb",
 * with the chroot as "./chroot/" for example.
 */
```

- Failing to drop root privileges required for invoking chroot(). After creating the chroot-jail, the elevated privileges should be dropped in order to satisfy the principle of least privilege. Listing 7.17 illustrates an error in shadow-4.1.2.2 where the root privileges are never dropped after the call to chroot() and chdir().

Listing 7.17: Failure to drop root privileges after chroot() in shadow-4.1.2.2

```
void subsystem (const struct passwd *pw){
    /*...*/
    if (pw->pw_dir[0] != '/') { /*...*/
        /*...*/
        if (chdir (pw->pw_dir) || chroot (pw->pw_dir)) {
            printf (_("Can't change root directory to '%s'\n"),
                    pw->pw_dir);
            SYSLOG ((LOG_WARN, NO_SUBROOT2, pw->pw_dir, pw->pw_name));
            /*...*/
        }
    }
}
```

## 7.4 Experiments in Data-Driven Mode

In the second step of the tool experimentation, we enabled the data dependency analysis. Table 31 shows the analysis details of packages freeradius-server-2.1.3, shadow-4.1.1, and kstart-3.14. The verification time and the detected errors are given for each executable in both control-flow mode analysis and data-driven mode analysis. The Remopla data dependency handling naturally leads to longer

**Table 31** Results of the experiments in the data-driven mode

Package	Property	Program	Model Checking (Sec)			Finding	
			CFG	DFA	Slowdown	CFG	DFA
freeradius-server 2.1.3	Null Check	radadmin	0.28	0.46	1.64	0	0
		radsniff	0.17	0.5	2.94	0	0
		radconf2xml	0.36	0.5	1.39	0	0
		radclient	0.41	4.7	11.46	0	0
		radwho	0.12	0.48	4	0	0
		radeapclient	0.61	6	9.8	2	1
		radiusd	0.34	0.74	2.18	0	0
kstart-3.14	Unsafe Env	k5start	0.3	0.73	2.43	0	0
		krenew	0.34	3.65	10.74	1	2
		k4start	0.2	5.54	27.7	1	2
shadow-4.1.2.2	Unsafe Env	su	15.5	70	4.52	0	1
		login	16.9	57.2	3.38	0	0
		groupmems	17.2	40.4	2.34	0	0
		useradd	13.8	37.2	2.69	0	0

verification time. The slowdown factor given in the sixth column of Table 31 indicates the performance overhead induced by the data dependency analysis. However, the data dependency awareness of our approach renders it more precise and efficient than MOPS. In fact, MOPS does not handle aliasing neither parameter passing during program verification as detailed in Chapter 3.

We show in the following paragraphs the errors that we detect in some software packages when enabling the data dependency analysis.

### 7.4.1 Parameter Passing

The sample code in Listing 7.18 is taken from the `kstart-3.14` package. Variable `aklog` is assigned the unsafe return value from `getenv()`. The content of `aklog` is not validated before being passed to the security critical function `system()`. The

only check performed is a null checking of `aklog`. Note that our pattern matching approach involves syntactic matching and scope matching of variables. Thus, the two `aklog` variables involved in this error do not match since they are in two different scopes.

Listing 7.18: Unsafe usage of environment variable in `kstart-3.14`

```
void command_run(const char *aklog, int verbose){
    int status;
    status = system(aklog);
    /*...*/
}
/*...*/
int main (int argc, char * argv[]){
    /*...*/
    aklog = getenv("AKLOG");
    if (aklog == NULL)
        aklog = getenv("KINIT_PROG");
    if (aklog == NULL)
        aklog = PATH_AKLOG;
    /*...*/
    /* If requested, run the aklog program. */
    if (do_aklog)
        command_run(aklog, verbose);
    /*...*/
}
```

The analysis performed in control-flow mode, without considering parameter passing, is not able to capture this error. When enabling data-driven analysis, the dependency between the `aklog` variables is captured by handling the parameter passing of function `command_run()`. As a result, the untrustworthy source of `aklog` used in `system()` call is captured.

### 7.4.2 Variable Aliasing

Listing 7.19 illustrates an unsafe usage of environment variables in program `su` of package `shadow-4.1.1`. This error cannot be detected by our analysis in control-flow mode. It cannot neither be captured by MOPS since it does not handle aliasing relations of variables.

Listing 7.19: Unsafe usage of environment variable in shadow-4.1.1

```

int main (int argc, char **argv)
{
    char *cp;
    const char *tty = 0; /* Name of tty SU is run from*/

    if (isatty (0) && (cp = ttyname (0))) {
        /*...*/
        tty = cp;
    }
    #ifndef USE_PAM
        is_console = console (tty);
    #endif
}
int console (const char *tty)
{
    return is_listed ("CONSOLE", tty, 1);
}
static int
is_listed (const char *cfgin,
           const char *tty, int def)
{
    FILE *fp;
    char buf[200], *cons, *s;
    /*...*/
    while ((s = strtok (cons, ":")) != NULL) {
        if (strcmp (s, tty) == 0)
            return 1;
    }
    /*...*/
}

```

The error trace starts in the main() function when variable cp is assigned the unsafe return value of the environment function ttyname(). The analysis in data-driven mode generates an aliasing relation out of the assignment operation tty = cp;. Besides, the handling of parameter passing allows our approach to track the origin of the actual argument tty passed to function console. Variable tty is passed again to function is\_listed where it is used as a parameter to function strcmp(). Since tty is derived from function ttyname, it is not guaranteed to be null-terminated and cannot be safely passed to strcmp(). This error trace is tricky since it involves two levels of function parameter passing and an aliasing relation between security-relevant variables. Uncovering this error demonstrates the efficiency, the precision, and the scalability features of our tool.

### 7.4.3 Reducing False Positives

Tracking data dependencies not only helps to detect data-driven defects, but also contributes in eliminating false positives. For example, in Listing 7.20, the variable `buffer` is a newly allocated pointer. Then, pointer `buffer` is passed to pointer `hdr`, which is checked against `NULL`, preventing the use of a null pointer. The control-flow mode analysis cannot locate the null check of `buffer` and hence reports a violation of the null check property. On the other hand, the data-driven mode analysis captures the dependency between pointers `buffer` and `hdr` and would not produce a false alarm.

Listing 7.20: False positive of unchecked return value `freeradius-server-2.1.3`

```
int
map_eapsim_basictypes(RADIUS_PACKET *r,
                     EAP_PACKET *ep){
    buffer = (unsigned char *)malloc(hmaclen);
    hdr = (eap_packet_t *)buffer;
    if (!hdr)
    {
        /*...*/
        return 0;
    }
    /*...*/
    fr_hmac_sha1(buffer, hmaclen,
                vp->vp_octets, vp->length, sha1digest);
    /*...*/
}
```

Our experimental results demonstrate that considering data dependencies brings precision at a significant performance cost. Like all static verification tools, we face a trade-off between scalability and precision of the analysis. Our security verification tool keeps the choice between control-flow mode and data-driven mode at users discretion to better fit their needs.

## 7.5 Comparison with existing tools

This section compares our security verification framework with existing tools. As presented in Chapter 3 MOPS is a pushdown model-checking tool for C programs. It has been successful in detecting programming errors in Linux kernel. The control-flow mode of our tool is similar to MOPS: we can detect the same error MOPS detects in almost the same time frame. Though the data dependency awareness of our approach renders it more precise and efficient than MOPS. In fact, MOPS does not handle aliasing neither parameter passing during program verification. Beside, MOPS has been designed and implemented for exclusively handling C language. our approach benefits from the GIMPLE representation in order to be extended to all languages that GCC compiles. MetaCompilation (MC) is a static analysis tool that uses a flow-based analysis approach for detecting temporal security errors in C code [11]. With the MC approach, programmers define their temporal security properties as automata written in a high-level language called Metal [65] based on syntactic pattern matching. In our approach, we benefit from the expressiveness of the procedural Remopla language to achieve the same level of expressiveness of Metal. A key difference is that metal patterns reference the source code directly, whereas our patterns are closer to the compiler representation and reference GIMPLE constructs. Soundness is another important difference between our approach and MC approach. Our analysis is sound with respect to generated program model, whereas MC sacrifices soundness for the sake of scalability. BLAST [68], SAT [37] and SLAM [16] are data-flow sensitive

model-checkers based on predicate abstraction. They use an iterative refinement process to locate security violations in source code. Both are mainly used to verify small software of device drivers. Despite the precision of their approach, their iterative process introduces the risk of non-termination and does not scale to large software. GMC2 [64] is a model-checker for the GCC compiler. As we do, GMC2 takes advantage of the TREE-SSA framework and its GIMPLE intermediate representation to tackle open source software. Nevertheless, GMC2 has not been used to verify large scale C software as we did.

## 7.6 Conclusion

In this chapter, we have detailed the experiments on large scale C software conducted with our security verification framework. First, we run our tool in the control-flow mode. Then, we activated the data-driven mode to enhance the precision analysis. We detected errors that other tools such MOPS cannot detect since they do not take into account data dependencies. There are two main sources of false positives generated with our tool:

- The verification process is path-insensitive and may report an error related to an infeasible trace. This limitation applies to the control-flow mode and the data-driven mode of our tool.
- For the control-flow mode, the false positive may be related to relevant data information that is not taken into account. We showed in the experimentation

results how our data-driven mode can reduce these kinds of false positives.

- In addition, our tool does not consider runtime flow information that is not captured in the compiler generated control-flow graph. As such, we do not handle long jumps (`setjmp()/longjmp()`) indirect calls via function pointer, multi-threading with signal handlers.

The experimentation results demonstrate the efficiency and the usability of our tool in detecting real errors in real-software packages. The integration of the CERT coding rules in our framework renders it a practical tool for assisting programmers in building secure software compliant with the CERT secure coding standard.



# Chapter 8

## Conclusion

This chapter concludes our thesis. First, we give a summary of our contributions, then we describe the research directions that can be performed in the future as an extension to our work.

### 8.1 Summary

Growing assurance requirements for applications and systems have raised the stakes on software safety and security. Software development process should take into account safety and security attributes at early stages. A special emphasis should be put on the implementation phase, since the root cause of many security vulnerabilities are programming errors that may yield readily exploitable code. As the size and the complexity of software increase, manual code review becomes an expensive and difficult challenge. Programmers need automated tools to assist them detecting

vulnerabilities in their code for the purpose of fixing them. In this thesis, we have elaborated approaches and techniques for the automated detection of safety and security violations in source code. We tackled safety properties to ensure that programs are free from our targeted set of type and memory errors. We also considered the violations of system-specific security properties that we refer to as high-level security properties.

### 8.1.1 Type and Effect Discipline for C Safety

We described our type and effect discipline for the detection of memory and type errors in C programs. We extended the standard C type system with safety annotations and static checks to uncover unsafe memory and type operations. We described an annotation inference algorithm that propagates annotations to program expressions and applies static checks for safety error detection. Our type and effect analysis has a number of appealing properties that we describe hereafter:

- **Simplicity:** the inference algorithm automatically propagates lightweight annotations and releases programmers from the cumbersome burden of manual annotations.
- **Effectiveness:** the flow-sensitivity and alias-sensitivity of our analysis enhance its efficiency for uncovering insidious errors.
- **Flexibility:** the type analysis can easily be combined with dynamic verification techniques in order to increase the precision of the overall approach.

- Usability: the prototype of our safety analysis is designed and implemented as an extension of the GCC compiler. The intent was to demonstrate that our analysis can be integrated within the compilation process. The safety analysis is triggered by simply flagging an option when invoking GCC.

In addition, we established that our static analysis captures all occurrences of our targeted set of memory and type errors. To this end, we described an operational semantics of our C-like language that complies with the ANSI C standard. Besides, the semantics evaluates undefined behaviors of memory and type operations to runtime errors. We proved the consistency of our static semantics and operational semantics. Based on the consistency results, we established the soundness of our analysis in detecting memory and type errors.

### 8.1.2 Automatic Verification of Security Properties

For the verification of system-specific security properties, we described our security verification environment that combines static analysis and model-checking. The combination of these two approaches consists in utilizing static analysis to automatically build a model-checkable abstraction of programs. It also takes advantage of the model-checking flexibility in verifying a wide range of system-specific security properties. The latter are modeled as finite state automata where nodes represent program states and transitions syntactically match program actions and expressions. Our implementation is based on the GIMPLE intermediate representation of the GCC compiler and the off-the-shelf model-checker for pushdown systems, namely Moped.

Our tool performs security verification in two modes: a control-flow mode that discards data dependencies and a data-driven mode that computes data dependencies between program expressions. The main features of our security verification tool are described hereafter:

- Our conducted experiments showed that our approach can be applied to large software projects. We used our tool to model-check a set of real world software. We were also able to catch real errors in the analyzed packages.
- The data-driven mode allows our approach to cope with aliasing and parameter passing that cannot be resolved using simplistic pattern matching approaches.
- The simplicity and the expressiveness of the GIMPLE representation enable us to increase the precision of our analysis and to have access to valuable environment information generated by the compiler.

## 8.2 Future Work

The work presented in this thesis can be extended in different directions. We provide hereafter the future extension plan of our work:

- Enlarge the set of coding errors that our type and effect discipline targets. By defining additional safety annotations, our type analysis can be used to detect a larger set of coding errors such as buffer overflows, format string errors, and input validation errors.

- Augment the high-level security verification tool with additional languages that GCC supports. For now, our approach focuses essentially on the C programming language. Nevertheless, we based our verification framework on the language-independent and platform-independent GIMPLE representation of the GCC compiler. Thus, our approach has the appealing extension feature to support all languages that GCC compiles.
- Interact with security hardening approaches to fix the detected coding errors. Our framework can be extended to provide an end-to-end solution for programmers that takes as input a source code, detects its security and safety violations, fix the detected errors, and outputs a security hardened source code.

## List of Publications

### Book Chapter

1. M-A. Laverdire, A. Mourad, S. Tlili and M. Debbabi. *Middleware Security in Wireless Applications*. In the *Encyclopedia of Wireless and Mobile Communications Book*, CRC Press, 2008, Taylor & Francis Group

### Journal Paper

1. S. Tlili and M. Debbabi. *Interprocedural and flow-sensitive type analysis for memory and type safety of C code*. *Journal of Automated Reasoning*, 42(2-4):265-300, 2009, Springer.

## Conference Papers

1. R. Hadjidj, X. Yang, S. Tlili, M. Debbabi. Model-Checking for Software Vulnerabilities Detection with Multi-Language Support. Proceedings of the sixth Annual Conference on Privacy, Security and Trust, PST 2008, Fredericton, New Brunswick, Canada, IEEE.
2. S. Tlili, Z. Yang, H. Ling, M. Debbabi. A Hybrid Approach for Safe Memory Management in C. Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, AMAST 2008, Urbana, IL, Springer Press.
3. S. Tlili and M. Debbabi. Type and Effect Annotations for Safe Memory Access in C. Proceedings of the 3rd International Conference on Availability, Reliability and Security, ARES'2008, Barcelona, Spain, 2008, IEEE Press.
4. S. Tlili and M. Debbabi. Novel Flow-Sensitive Type and Effect Analysis for Securing C code. Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, AICCSA'08, Security and Information Assurance Track, IEEE Press.

# Bibliography

- [1] Build Security In, April 2009. <https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home.html>.
- [2] CERT Secure Coding Standards, April 2009. <http://www.securecoding.cert.org>.
- [3] Remopla introduction, April 2009. <http://www.fmi.uni-stuttgart.de/szs/tools/moped/remopla-intro.pdf>.
- [4] AbsInt. AbsInt: Advanced Compiler Technology for Embedded Systems. <http://www.absint.com/>.
- [5] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 230–246, London, UK, 2002. Springer-Verlag.

- [6] Ashish Aggarwal and Pankaj Jalote. Integrating Static and Dynamic Analysis for Detecting Vulnerabilities. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 343–350, Washington, DC, USA, 2006. IEEE Computer Society.
- [7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [8] Alexander Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An Overview of the Saturn Project. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, New York, NY, USA, 2007. ACM.
- [9] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.
- [10] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
- [11] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, Washington, DC, USA, 2002. IEEE Computer Society.



- [12] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of all Pointer and Array Access Errors. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, New York, NY, USA, 1994. ACM.
- [13] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving Software Security with a C Pointer Analysis. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 332–341, New York, NY, USA, 2005. ACM.
- [14] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 79–88, New York, NY, USA, 2008. ACM.
- [15] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, pages 203–213, New York, NY, USA, 2001. ACM.
- [16] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.
- [17] Thoms Bell. The Concept of Dynamic Analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th*

*ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234, London, UK, 1999. Springer-Verlag.

- [18] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, 2007.
- [19] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 2(2):131–152, 1996.
- [20] Matt Bishop. How Attackers Break Programs, and How to Write More Secure Programs.
- [21] Matt Bishop. How to Write a Setuid Program. Technical Report Technical Report 85.6, Research Institute for Advanced Computer Science, Moffett Field, May 1985.
- [22] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, Inc., 3 edition, November 2005.
- [23] Guillaume Brat and Willem Visser. Combining static analysis and model checking for software analysis. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 262, Washington, DC, USA, 2001. IEEE Computer Society.
- [24] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *Logic in Computer*

- Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e,*  
pages 428–439, 1990.
- [25] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-Insensitve Interprocedural Alias Analysis in the Presence of Pointers. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250, London, UK, 1995. Springer-Verlag.
- [26] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.
- [27] John P. McDermott Carl E. Landwehr, Alan R. Bull and William S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Comput. Surv.*, 26(3):211–254, 1994.
- [28] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *in Proceedings of FMCAD*, pages 33–51, 2002.
- [29] Benjamin Chelf, Dawson Engler, and Seth Hallem. How to Write System-Specific, Static Checkers in Metal. *SIGSOFT Softw. Eng. Notes*, 28(1):51–60, 2003.
- [30] Hao Chen, Drew Dean, and David Wagner. Model Checking One Million Lines of C Code. In *NDSS*, 2004.

- [31] Hao Chen and David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, Washington, DC, november 2002.
- [32] Hao Chen and David A. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, New York, NY, USA, 2002. ACM.
- [33] Hao Chen and David A. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. Technical Report UCB/CSD-02-1197, EECS Department, University of California, Berkeley, 2002.
- [34] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 232–245, New York, NY, USA, 1993. ACM.
- [35] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive program design*, pages 305–349, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

- [36] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
- [37] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.
- [38] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, 2001. Springer-Verlag.
- [39] Jacques Corbin and Michel Bidoit. A Rehabilitation of Robinson’s Unification Algorithm. In *IFIP Congress*, pages 909–914, 1983.
- [40] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
- [41] P. Cousot and R. Cousot. Parallel combination of abstract interpretation and model-based automatic analysis of software. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software, AAS’97*, pages 91–98, Paris, France, January 1997. ACM Press, New York, New York, United State’s.

- [42] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engg.*, 6(1):69–95, 1999.
- [43] Coverity. Coverity Prevent for C and C++. <http://www.coverity.com/main.html>.
- [44] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
- [45] Manuvir Das. Unification-based pointer analysis with directional assignments. *SIGPLAN Not.*, 35(5):35–46, 2000.
- [46] Mourad Debbabi, Zahia Aidoud, and Ali Faour. On the Inference of Structured Recursive Effects with Subtyping. *Journal of Functional and Logic Programming*, 1997(5), 1997.
- [47] Takahiro Shinagawa Dept. Implementing A Secure Setuid Program.
- [48] Dawson Engler. Static Analysis versus Model Checking for Bug Finding. pages 1–1, 2005.
- [49] Michael D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.
- [50] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In E. Allen Emerson and

- A. Prasad Sistla, editors, *Proceedings of CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, July 2000.
- [51] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:27–56, June 2008. Special Issue on Constraints to Formal Verification.
- [52] David Evans. Static Detection of Dynamic Memory Errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, pages 44–53, New York, NY, USA, 1996. ACM.
- [53] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19(5):87–96, 1994.
- [54] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *Symposium on the Foundations of Software Engineering*, December 1994.
- [55] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 13–24, New York, NY, USA, 2002. ACM.

- [56] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. *SIGPLAN Not.*, 35(5):219–232, 2000.
- [57] International Organization for Standardization (ISO) and International Electrotechnical Commission. Information Technology – Programming languages, Their Environments and System Software Interfaces – Specification for Safer, More Secure C Library Functions. Technical Report ISO/IEC TR 24731-1:2006, International Organization for Standardization, 2006.
- [58] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *SIGPLAN conference on Programming Language Design and Implementation*, pages 192–203, 1999.
- [59] The FreeBSD Foundation. FreeBSD the power to serve. <http://www.freebsd.org/>.
- [60] Frances E. Allen. Control Flow Analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.
- [61] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.
- [62] D. Goyal. An improved intra-procedural may-alias analysis algorithm. Technical report, New York, NY, USA, 1999.



- [63] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 conference on Programming Language Design and Implementation*, pages 282–293, New York, NY, USA, 2002. ACM.
- [64] R. Grosu, X. Huang, S. Jain, and S. A. Smolka. Open source model checking. In *In Proceedings of the Workshop on Software Model Checking, Edinburgh, Scotland, July*, pages 27–44. Elsevier, 2005.
- [65] Seth Hallett, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-Specific, Static Analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, New York, NY, USA, 2002. ACM.
- [66] R. Hasting and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, January 2002.
- [67] Matthew Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
- [68] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 58–70, New York, NY, USA, 2002. ACM.

- [69] Tomoyuki Higuchi and Atsushi Ohori. A Static Type System for JVM Access Control. In *ICFP*, pages 227–237, 2003.
- [70] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM.
- [71] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
- [72] Susan Horwitz. Debugging via run-time type checking. *SIGSOFT Softw. Eng. Notes*, 25(1):58, 2000.
- [73] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.
- [74] Linux Online Inc. Linux Online! <http://www.linux.org/>.
- [75] ISO. The ansi c standard (c99). Technical Report WG14 N1124, ISO/IEC, 1999. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf>.
- [76] ISO. *ISO/IEC 14882:2003: Programming languages: C++*. 2003. <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110>.
- [77] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 conference on*

*Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2002. ACM.

- [78] Pavel Jezek, Jan Kofron, and Frantisek Plasil. Model checking of component behavior specification: A real life experience. *Electr. Notes Theor. Comput. Sci.*, 160:197–210, 2006.
- [79] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 119–134, Berkeley, CA, USA, 2004. USENIX Association.
- [80] Stephen Johnson. Lint, a C program checker. Technical report, Bell Laboratories, Computer Science Technical Report 65, 1977.
- [81] Assaf J. Kfoury, S. Ronchi della Rocca, Jerzy Tiuryn, and Pawel Urzyezytn. Alpha-Conversion and Typability. *Information and Computation*, 150(1):1–21, 1999.
- [82] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
- [83] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. *SIGPLAN Not.*, 28(6):56–67, 1993.

- [84] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *10th USENIX Security Symposium*, pages 177–190. University of Virginia, Department of Computer Science, USENIX Association, August 2001.
- [85] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.
- [86] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Softw. Pract. Exper.*, 24(2):197–218, 1994.
- [87] Roger E. Lessman. Changes and Extensions in the C Family of Languages. *SIGCSE Bull.*, 21(2):34–39, 1989.
- [88] Peng Li. Safe systems programming languages, 2004.
- [89] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for Evaluating Bug Detection Tools. In *Work. on the Evaluation of Software Defect Detection Tools*, June 2005.
- [90] Neil Matthew, Richard Stones, and Alan Cox. *Beginning Linux Programming, Third Edition*. Wrox Press Ltd., Birmingham, UK, UK, 2003.
- [91] Bertrand Meyer. *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Publishing Company, Incorporated, 2009.

- [92] Robin Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
- [93] Michael Ernst Mit and Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *In WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.
- [94] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
- [95] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
- [96] Flemming Nielson and Hanne Riis Nielson. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances*, pages 114–136, London, UK, 1999. Springer-Verlag.
- [97] Diego Novillo. Tree-SSA: A New Optimization Infrastructure for GCC. In *Proceedings of the GCC Developers Summit3*, pages 181–193, Ottawa, Ontario, Canada, 2003.
- [98] Yoann Padiou, Lin Tan, and Yuanyuan Zhou. Listening to programmers - Taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering (ICSE09)*, May 2009.

- [99] Parasoft Insure++. We make software work. <http://www.parasoft.com/>.
- [100] James W. Paulson, Giancarlo Succi, and Armin Eberlein. An empirical study of open-source and closed-source software products. *IEEE Trans. Softw. Eng.*, 30(4):246–256, 2004.
- [101] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- [102] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
- [103] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
- [104] PolySpace. Automatic Detection of Run-Time Errors at Compile Time.
- [105] Corneliu Popeea, Dana N. Xu, and Wei-Ngan Chin. A Practical and Precise Inference and Specializer for Array Bound Checks Elimination. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial Evaluation and Program Manipulation*, pages 177–187, New York, NY, USA, 2008. ACM.
- [106] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

- [107] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted push-down systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2):206–263, October 2005. Special Issue on the Static Analysis Symposium 2003.
- [108] John A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.
- [109] Radu Rugina and Sigmund Chorem. Region Inference for Imperative Languages. Technical Report CS TR2003-1914, Computer Science Department, Cornell University, 2003.
- [110] Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Program Analysis Using Symbolic Ranges. In *SAS '07: Proceedings of the 14th International Static Analysis Symposium*, pages 366–383, Kongens Lyngby, Denmark, 2007. Springer.
- [111] Benjamin Schwarz, Hao Chen, David A. Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model Checking an Entire Linux Distribution for Security Violations. In *Proceedings of the 2005 Annual Computer Security Applications Conference (ACSAC)*, 2005.
- [112] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

- [113] Robert Seacord. Secure coding in c and c++: Of strings and integers. *IEEE Security and Privacy*, 4(1):74, 2006.
- [114] Helmut Seidl and Christian Fecht. Interprocedural analysis based on pdas. *Universität Trier, Mathematik/Informatik, Forschungsbericht*, 97-06, 1997.
- [115] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, pages 17–30, Anaheim, California, USA, April 2005.
- [116] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium, 2001.*, pages 201–220, 2001.
- [117] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with Type Casts in C. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 180–198, London, UK, 1999. Springer-Verlag.
- [118] Fortify Software. Rats - rough auditing tool for security, April 2009. <http://www.fortify.com/security-resources/rats.jsp>.
- [119] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 32–41, New York, NY, USA, 1996. ACM.



- [120] Martin Franz Sulzmann. *A general framework for hindley/milner type systems with constraints*. PhD thesis, New Haven, CT, USA, 2000. Director-Hudak, Paul.
- [121] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2:245–271, 1992.
- [122] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Information and Computation*, pages 162–173, 1992.
- [123] Syrine Tlili, Zhenrong Yang, Hai Zhou Ling, and Mourad Debbabi. A Hybrid Approach for Safe Memory Management in C. In *AMAST'08: Proceedings of the 12th international conference on Algebraic Methodology and Software Technology*, pages 377–391, Urbana, Illinois, USA, 2008. Springer-Verlag.
- [124] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988.
- [125] Twitch. Taking Advantage of Non-Terminated Adjacent Memory Spaces. *Phrack*, 56, May 2000.
- [126] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257. IEEE Computer Society, 2000.

- [127] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated Software Engineering*, pages 3–12, Washington, DC, USA, 2000. IEEE Computer Society.
- [128] Gray Watson. Debug Malloc Library, October 2004. <http://dmalloc.com/>.
- [129] David A. Wheeler. Secure Programming for Linux and Unix HOWTO, 2003.
- [130] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 1995. ACM.
- [131] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.
- [132] Karen Yorav, editor. *Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007, Proceedings*, volume 4899 of *Lecture Notes in Computer Science*. Springer, 2008.
- [133] Y. Younan, W. Joosen, and F. Piessens. Code Injection in C and C++ : A Survey of Vulnerabilities and Survey of Vulnerabilities and Countermeasures.

Technical Report CW386, Department of Computer Science, Katholieke Universiteit Leuven, July 2004.

- [134] Oiwa Yutaka, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure: Progress Report. In *ISSS'02: Proceedings of the International Symposium of Software Security*, pages 133–153. Springer-Verlag, 2002.

# Appendices

## Appendix I: Static Analysis Utility Functions

We define in the current appendix the auxiliary functions used in the algorithms of Chapter 4.

- Function `regionOf` takes a pointer type and returns its region annotations.  
`regionOf`: *Inferred Types*  $\rightarrow$  *Regions*

```
Function regionOf( $\tau$ ) = case  $\tau$  of  
   $ref_{\rho}(\kappa)_{\eta}$   $\Rightarrow$   $\rho$   
   $if(\tau, \tau')$   $\Rightarrow$  regionOf( $\tau$ )  $\cup$  regionOf( $\tau'$ )  
  else  $\Rightarrow$   $\emptyset$   
end
```

- Function `rootOf` takes as argument an lvalue and returns its related variable.  
`rootOf`: *Lval*  $\rightarrow$  *Lval*

```
Function rootOf( $l_v$ ) = case  $l_v$  of  
   $x$   $\Rightarrow$   $x$   
   $*l'_v \mid l'_v.\varphi$   $\Rightarrow$  rootOf( $l'_v$ )  
end
```

- Function `addressOf` returns the memory location of an lvalue argument.  
`addressOf`: *Lval*  $\times$  *Env*  $\rightarrow$  *Regions*

**Function**  $\text{addressOf}(l_v, \mathcal{E}) =$   
 let  $x = \text{rootOf}(l_v)$   
      $\tau = \mathcal{E}(x)$   
 in  
 case  $(l_v, \tau)$  of  
    $(x, \tau) \Rightarrow r_x$   
    $(*x, \text{ref}_\rho(\_)\_ ) \Rightarrow \rho$   
    $(x.\varphi, \tau) \Rightarrow r_x.\text{offset}(\varphi)$   
    $(*l_v, \tau) \Rightarrow \text{addressOf}(l_v, \mathcal{E})$   
    $(*l_v.\varphi, \tau) \Rightarrow \text{addressOf}(l_v.\varphi, \mathcal{E})$   
 end

- Function  $\text{fldType}(\tau, \varphi_i)$  extracts the type of field  $\varphi_i$  from a structure type  $\tau$ .  
 $\text{fldType} : \text{Inferred Types} \times \text{Fields} \rightarrow \text{Inferred Types}$

**Function**  $\text{fldType}(\tau, \varphi_i) = \text{case } \tau \text{ of}$   
    $\text{if}(\tau', \tau'') \Rightarrow \text{if}(\text{fldType}(\tau', \varphi_i), \text{fldType}(\tau'', \varphi_i))$   
    $\text{struct}\{(\varphi_i, \tau_i, o_i)\}_{1..n} \Rightarrow \tau_i$   
 end

- Function  $\text{typeOf}$  returns the annotated type of an lvalue argument.  
 $\text{typeOf} : \text{Lval} \times \text{Env} \rightarrow \text{Inferred Types}$

**Function**  $\text{typeOf}(l_v, \mathcal{E}) =$   
 case  $l_v$  of  
    $x \Rightarrow \mathcal{E}(x)$   
    $*l_v \Rightarrow \text{strTypeOf}(\text{typeOf}(l_v, \mathcal{E}))$   
    $l_v.\varphi \Rightarrow \text{fldType}(\text{typeOf}(l_v, \mathcal{E}), \varphi)$   
 end

- Function  $\text{hostOf}$  takes a pointer type and returns its host annotations.  
 $\text{hostOf} : \text{Inferred Types} \rightarrow \text{Pointer Host}$

**Function**  $\text{hostOf}(\tau) = \text{case } \tau \text{ of}$   
    $\text{ref}_\rho(\kappa)_\eta \mid \text{int}_\eta \Rightarrow \eta$   
    $\text{if}(\tau, \tau') \Rightarrow \text{hostOf}(\tau) \cup \text{hostOf}(\tau')$   
   else  $\Rightarrow \emptyset$   
 end

- Function  $\text{updHost}$  takes a type and host annotation and returns a type.  
 $\text{updHost} : \text{Inferred Types} \times \text{Host Annotations} \rightarrow \text{Inferred Types}$

**Function**  $\text{updHost}(\tau, \eta) = \text{case } \tau \text{ of}$   
 $\text{if}(\tau', \tau'') \Rightarrow \text{if}(\text{updHost}(\tau', \eta), \text{updHost}(\tau'', \eta))$   
 $\text{int}_{\eta'} \Rightarrow \text{int}_{\eta}$   
 $\text{ref}_{\rho}(\_)_{\eta'} \Rightarrow \text{ref}_{\rho}(\_)_{\eta}$   
**end**

- Function  $\text{updRegHost}$  takes a type, a set of regions, and a host annotation, and returns a type.

$\text{updRegHost} : \text{Inferred Types} \times \text{Regions} \times \text{Host Annotations} \rightarrow \text{Inferred Types}$

**Function**  $\text{updRegHost}(\tau, \rho, \eta) = \text{case } \tau \text{ of}$   
 $\text{if}(\tau', \tau'') \Rightarrow \text{if}(\text{updRegHost}(\tau', \rho, \eta), \text{updRegHost}(\tau'', \rho, \eta))$   
 $\text{ref}_{\rho'}(\kappa)_{\eta'} \Rightarrow \text{if } (\rho \cap \rho' \neq \emptyset) \text{ then}$   
 $\quad \text{ref}_{\rho'}(\kappa)_{\eta}$   
 $\quad \text{else}$   
 $\quad \tau$   
**end**

- Function  $\text{regHostOf}$  takes a type and returns a set of region and host annotation pairs.

$\text{regHostOf} : \text{Inferred Types} \rightarrow \mathcal{P}(\text{Regions} \times \text{Host Annotation})$

**Function**  $\text{regHostOf}(\tau) = \text{case } \tau \text{ of}$   
 $\text{if}(\tau', \tau'') \Rightarrow \text{regHostOf}(\tau') \cup \text{regHostOf}(\tau'')$   
 $\text{ref}_{\rho'}(\kappa)_{\eta'} \Rightarrow \{(\rho, \eta')\}$   
 $\text{else} \Rightarrow \text{fail}$   
**end**

- Function  $\text{updFld}$  takes a type, a field label, and another type, and returns a type.

$\text{updFld} : \text{Inferred Type} \times \text{Field} \times \text{Inferred Type} \rightarrow \text{Inferred Type}$

**Function**  $\text{updFld}(\tau, \varphi, \tau') = \text{case } \tau \text{ of}$   
 $\text{if}(\tau_1, \tau_2) \Rightarrow \text{if}(\text{updFld}(\tau_1, \varphi, \tau'), \text{updFld}(\tau_2, \varphi, \tau'))$   
 $\text{struct}\{(\tau_i, \varphi_i, o_i)\}_{1..n} \Rightarrow \text{struct}\{(\tau'_i, \varphi_i, o_i)\}_{1..n}$   
 $\quad \text{where } \begin{cases} \tau'_i = \tau' & \text{if } \varphi_i = \varphi \\ \tau'_i = \tau_i & \text{otherwise.} \end{cases}$   
**end**

- Function  $\text{RegSeqOf}$  takes a pointer type and returns its sequence of region annotations.

$\text{RegSeqOf} : \text{Inferred Types} \rightarrow \text{Sequence of Regions}$

The symbol  $+$  denotes the concatenation operator.

```
Function RegSeqOf( $\tau$ ) = case  $\tau$  of  
   $ref_{\rho}(\kappa)_{\eta}$   $\Rightarrow$  [ $\rho$ ]  
   $if(\tau, \tau')$   $\Rightarrow$  RegSeqOf( $\tau$ ) + RegSeqOf( $\tau'$ )  
  else  $\Rightarrow$   $\emptyset$   
end
```

- Function HostSeqOf takes a pointer type and returns its sequence of region annotations.

HostSeqOf: *Inferred Types*  $\rightarrow$  *Sequence of Host Annotations*

The symbol  $+$  denotes the concatenation operator.

```
Function HostSeqOf( $\tau$ ) = case  $\tau$  of  
   $ref_{\rho}(\kappa)_{\eta}$   $\Rightarrow$  [ $\eta$ ]  
   $int_{\eta}$   $\Rightarrow$  [ $\eta$ ]  
   $if(\tau, \tau')$   $\Rightarrow$  HostSeqOf( $\tau$ )@HostSeqOf( $\tau'$ )  
  else  $\Rightarrow$   $\emptyset$   
end
```