Tel Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
School of Computer Science

Automatic Verification of Program Cleanness

by

Nurit Dor

under the supervision of Dr. Mooly Sagiv
and the consultation of Prof. Michael Rodeh

A thesis submitted
for the degree of Doctor of Philosophy

Submitted to the Senate of Tel Aviv University
December 2003

To Noa.

Abstract

Automatic Verification of Program Cleanness

Nurit Dor
Doctor of Philosophy
School of Computer Science
Tel Aviv University

The area of software quality is a rapidly growing source of concern as our daily life becomes increasingly dependent on software. Users expect applications to be secure, protected and reliable. However, we often learn about new security vulnerabilities that has caused loss of time, money, and which sometimes penetrates the privacy of users.

Many of the software defects result from misusing pointers, arrays, and standard functions (e.g., `gets(), strcpy(), malloc()` in C). These bugs are referred to as *cleanness* violations, since they should not exist in any "reasonable" program, independent of the program specification. Some cleanness conditions are defined by the programming language semantics, such as access to `NULL` pointer and buffer overflow (out of bound access). Others are regarded as good programming styles, e.g., absence of memory leakage.

Runtime checking can be effective in localizing the source of a violation — where a `NULL` pointer is dereferenced or where leaking memory is created. The major disadvantage of runtime checking tools is that they depend on the test data. Therefore, they cannot assure us against cleanness violations.

This thesis presents *static* checking algorithms that analyze the source code of programs, assume all inputs are possible and identify potential cleanness violations. Furthermore, the algorithms are *conservative*. i.e., do not overlook any cleanness violation. Thus, they can provide the assurance against cleanness violations.

Acknowledgements

First, I would like to thank my advisor, Mooly Sagiv. I am grateful for his teaching, dedicated supervision, guidance and for his moral support throughout my graduate studies. His vast knowledge, scientific methods and enthusiasm has contributed not only to this thesis but also to my development as a researcher. I have also been fortunate to have Michael Rodeh as a consultant. I am thankful for his guidance, involvement, and mostly for his inspiration. All of those are reflected in every milestone of my doctorate research.

I am thankful to Manuvir Das for being a guiding mentor during an interesting and contributive internship at Microsoft Research and for providing and assisting me with AST-ToolKit and GOLF. Special thanks go to others who have helped: Bertrand Jeannet and Nicolas Halbwachs for providing and supporting the polyhedra library, Greta Yorsh and Alex Warshavsky for their assistance in the prototype implementations and for numerous technical insights. Thanks to Jean Souyris for providing and assisting with Airbus code. I thank those who commented on earlier drafts of this thesis as well as on papers upon which this thesis is based: Rastislav Bodik, David Evans, John Field, Seth Hallem, Anat Lotan, Roman Manevich, G. Ramalingam, Tom Reps, Noam Rinetskey, Ran Shaham, Reinhard Wilhelm, Eran Yahav, and the anonymous referees.

Finally, I thank my beloved family: Amit for the never-ending encouragement, Ittai for all the beautiful moments that strengthened me, and Noa, who by surmounting tremendous obstacles gave me the perseverance needed to complete this work. This thesis is dedicated to her.

This research was supported in part by grants from the Israeli Ministry of Science, by the RTD project IST-1999-20527 “DAEDALUS” of the European FP5 programme, and by Check Point Software Technologies Ltd.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many software defects result from misusing pointers, arrays, and standard functions (e.g. `gets()`, `strcpy()`, `malloc()` in C). These bugs are referred to as *cleanness* violations, since they should not exist in any "reasonable" program, independent of the program specification. Furthermore, this type of bug may lead to security vulnerability. Good programming styles are also defined as cleanness rules. For example, absence of memory leakage or absence of aliasing among actual parameters.

In particular, *buffer overflows* in C have been causing serious security problems for decades. The most famous example, the Internet worm of 1988 used a buffer overflow in `fingerd` to exploit tens of thousands of machines and disrupted normal activities and Internet connectivity for many days all around the world [65]. The CERT/Coordination Center advisories show that the problem is growing instead of shrinking [68].

The wide number of safety violations is also indicated by the FUZZ study which shows that in 1990 a random input generator was able to crash 25–33% of the utilities programs in UNIX systems. In 1995 using the same approach, 15–43% of the utilities in commercial UNIX systems crashed or hanged (infinite loop) [47]. About 60% of the reported failures in the 1995 FUZZ study are due to runtime string-manipulation errors, such as buffer overflow, access beyond the bounds of a string and misuse of the null-termination byte.

This thesis concentrates on static algorithms to detect safety and cleanness violations. Unlike runtime techniques, static analysis does not depend on the input. Furthermore, the *conservative* approach takes into account all possible inputs and behaviors, and thus does not miss any violation. In this thesis, we design conservative static algorithms that can prove the absence of certain safety and cleanness violations in C programs with minimum *false alarms* (also called *false negatives*). A false alarm is a reported error that can never occur at runtime. By proving that a program is clean we achieve a code quality assurance guaranteeing that the program does not contain any unexpected behavior such as core dump or an unhandled exception. Thus, the behavior of the program is defined on all inputs. Other benefits, which are beyond the scope of this thesis, are optimization possibilities such as removing redundant runtime checks of array accesses.

We choose the C programming language since it is one of the most general languages. Furthermore, many academic and research projects aim at improving the reliability of C programs. However, little attention has been paid to the static analysis approach. We were challenged by the estimation that static analysis can provide efficient solutions to overcome the lack of security in C programs, and in particular the widely exploited buffer overflow vulnerability.

1.1 String Analysis

The C programming language has been designed for flexibility, generality and mostly for efficiency. Restrictions and safety checks are usually omitted and left for the programmer. *Strings* are frequently used in C programs, despite being one of the most unsafe types. Many of the standard library functions (e.g., `gets(), strcpy()`) are inherently unsafe. The programmer is responsible for checking that basic operations cannot *overflow* the buffer, i.e., writing beyond the bounds of the array. A string in C is an array of characters ending with the special null-termination byte ('\0'). Again, it is up to the programmer to ensure the existence of the null-termination byte within the bounds of the array. This is a tricky task since some functions have a misleading behavior. For example, the `strncpy()` function may leave the target buffer without a null-termination character. It is not surprising that many programs suffer from string violations such as access beyond the bounds, misuse of the null-termination byte and unsafe calls to standard library functions.

1.1.1 Overview

Although the problem of string manipulation safety checking is to verify that accesses are within bounds [41, 5, 56, 16], the domain of string programs requires that the analysis be capable of:

- handling standard C functions, such as `strcpy()` and `strlen()`, which perform an unbounded number of loop iterations
- statically estimating the length of strings (in addition to the sizes of allocated base addresses) which is dynamically changed based on the index of the first null-termination character
- simultaneously analyzing pointer and integer values in order to precisely handle pointer arithmetic and destructive updates.

Our goal is to perform static analysis that detects *all* string runtime errors with just a few false alarms. This goal is ambitious. Due to features of C such as pointer arithmetic, destructive updates through pointers, and dynamic allocation it is not trivial to show the absence of string violations. For example, to verify a simple and common statement such as `strcpy(dst,src)` one must conclude that `src` points-to a null-terminated string and that the allocation space from the location pointed-to by `dst` is large enough to contain the string pointed-to by `src`. Indeed, existing methods either miss errors, yield many false alarms or cannot handle complicated aspects of C, such as multilevel pointers and structures.

1.1.2 Contributions

This thesis contains two string analyses algorithms. We summarize the contribution of them as follows:

1. **S**tring **A**nalysis for **S**ubC (*SAS* for short) handles a subset of C and is presented in [20]. The main contributions of this work are:
   (a) A source-to-source transformation that produces an instrumented C program which asserts when a string-manipulation cleanness violation occurs.
   (b) We show that the integer analysis algorithm of Cousot and Halbwachs [12] can be used to analyze the instrumented program to find all potentially unclean string expressions in a rather precise way.
   (c) We have implemented both the source-to-source transformation and the static analysis algorithms and applied them to real C programs. The experimental results are quite encouraging. We analyze real programs and locate nontrivial software defects with only a few false alarms.

2. **C** **S**tring **S**tatic **V**erifyer (*CSSV* for short) is a tool that demonstrates that uncovering all string problems in an arbitrary C program is achievable. CSSV is capable of analyzing realistic procedures and produces rather precise results. We presented CSSV in an informal and more accessible manner in [21]. In this thesis CSSV is formalized and proven to be sound. The main contributions of this work are:

   (a) A conservative static-analysis algorithm for detecting string runtime errors is presented. The algorithm reduces the problem of checking string manipulation to that of checking integer manipulations—a problem for which well-known solutions exist. In comparison to the SAS algorithm, presented in [20], CSSV's algorithm handles the *full* spectrum of C language constructs, including dynamically allocated structures, multi-level arrays, multi-level pointers, function pointers, and casting. In addition, its complexity is an order of magnitude more efficient.

   (b) An algorithm that computes procedural pointer information from a given whole-program flow-insensitive pointer information is presented. The goal is to reduce the number of false alarms when analyzing well-behaved programs. Specifically, the algorithm can infer the following important properties: (i) that a formal parameter points to a single location throughout the procedure, (ii) that a local variable must be aliased to a formal parameter whenever it is initialized. Obtaining procedural pointer information is a fundamental problem that clients of whole-program flow-insensitive pointer information face. The problem stems from the fact that a formal parameter may point to different abstract locations. Thus, a naive implementation will perform weak updates which may lead to many false alarms. Hence, we believe that our algorithm can be used to improve the precision of other clients of whole-program flow-insensitive algorithms, such as slicing tools and program optimizers (e.g., [29, 44]).

   (c) We have implemented CSSV and applied it to real-life programs. CSSV verified an intricate string library from EADS Airbus yielding only six false alarms. In the application fixwrites, part of web2c, CSSV uncovered eight errors with two false alarms. We classify the reasons behind the false alarms. The false alarms in fixwrites are due to correctness relationships between two different procedures. These false alarms can be avoided only by adding redundant safety checks to the code. Fortunately, this is not the common case. The six false alarms in Airbus' code are due to imprecision in the analysis which takes into account infeasible states. To overcome theses false alarms we need to add an additional analysis which tracks values in arrays.

In comparing the two string analysis algorithms we show that CSSV is usually more precise and more efficient than the SAS algorithm. However, CSSV uses a flow-insensitive pointer analysis while SAS computes flow-sensitive information regarding pointer aliasing. Thus, for some programs the SAS algorithm is more accurate than the CSSV algorithm. We believe, based on our empirical results, that this is not the common case. Another potential advantage of the SAS algorithm is its simplicity.

1.1.3 Related Work

Many academic and commercial projects produce practical tools that detect string manipulation errors at runtime, e.g., [35, 3, 45, 13]. The main disadvantage of runtime checking is that its effectiveness strongly depends on the input tested, and it does not ensure against bugs arising on other inputs. Our goal is a conservative static tool that detects all string errors and provides an assurance against all such errors.

Many academic projects produce unsound tools to statically detect string manipulation errors. In [42] an extension to LCLint is presented. Unsound lightweight techniques, heuristics, and in-code annotations are employed to check for buffer overflow vulnerabilities. Eau claire [8], a tool based on ESC-Java, checks for security holes in C programs by translating a subset of C to guarded commands. It is also unsound and requires information about loops. In [68] Wagner et al. present an algorithm that statically identifies string errors by performing a flow insensitive unsound analysis. The main disadvantage of all of these unsound tools is that they miss errors while CSSV does not miss any error. Furthermore, none of these tools can track effects of pointer arithmetic, a widely used method for string manipulation. Our SAS algorithm is the first sound algorithm for statically detecting string errors. The work of Simon and King, presented in [64], follows our SAS algorithm and formalizes a string analysis under the framework of abstract interpretation. However, both our SAS work and the work of Simon and King cannot handle all C, in particular multi-level pointers and structures. CSSV is the first sound tool to handle all C and in a rather precise manner.

1.2 Linked List Analysis

Many nasty bugs result from misusing memory by `NULL` pointer dereference, access to freed storage, or memory leaks. No wonder that many academic and commercial projects are aimed at producing tools that detect this class of cleanness violations referred to as **memory cleanness violations**. This work is presented in [19].

1.2.1 Overview

We describe algorithms that automatically discover memory cleanness errors in C programs. The algorithms are conservative and based on pointer analysis techniques. We do not intend to introduce other shape analysis or pointer analysis algorithms. Instead, we focus on detecting cleanness violations by using such algorithms. We applied the algorithms to C programs which manipulate singly-linked lists.

1.2.2 Contributions

Two major results have been obtained:

- A new, efficient and quite precise algorithm for detecting memory cleanness errors. It does not yield **any** false alarm on quite intricate C programs. On programs manipulating more complicated data structures such as cyclic lists and trees, it produces false alarms. It represents memory states as Shape Graphs with Reachability from stack variables and thus we call it *SG+R*. To assess the usability of our new algorithm we implemented it using PAG – a program analyzer generator developed at Saarland University [46].

- We use two of the well known pointer analysis techniques to check cleanness and compare their cost and effectiveness to our new algorithm. The first technique is **flow-sensitive points-to** analysis [2, 62] (referred to as *PT*). The second algorithm, *SG*, uses Shape Graphs and is a slight variant of the shape analysis technique described in [58]. The *SG* algorithm is similar to *SG+R* except that it does not (explicitly) track reachability from stack variables. We compare time, space and the number of false alarms and conclude that:

  - As expected, the *PT* algorithm is the cheapest of the three but it is quite imprecise. In fact, it cannot be used at all to check for memory leakage errors.

– The *SG* algorithm is less effective than the *SG+R* algorithm and is sometimes slower. This is evidence to a general phenomenon in program analysis where a less precise algorithm can be sometimes less efficient in practice since the less precise algorithm may fail to detect the important program invariants, thereby abstracting situations that can never occur in the program and thus consuming more time and space.

1.2.3 Related Work

Dynamic Cleanness Checking

The vast majority of cleanness checking tools detect violations at run-time (e.g., see [3, 70, 54]). They can be effective in localizing the source of a violation — where a NULL pointer is dereferenced or where leaking memory is created. The Java run-time environment also checks for memory violations. However, it is clear that the effectiveness of run-time checking depends on the proper choice of test cases to uncover violations. Since run-time tools are intrusive, they are rarely used when the code is run in production. For example, the experimental results of Safe-C [3] show that execution overhead of cleanness checking ranges from 130% to 540%. These tools demand manual runs and therefore they do not ensure us against bugs.

Of course, any conservative tool including ours can also be used in conjunction with a dynamic tool to reduce the cost of run-time checks. The static tool can indicate which run-time checks are redundant. For example, the absence of memory leakage implies that garbage collection need not be invoked.

Static Cleanness Checking

The *LCLint* tool can discover certain cleanness violations by statically analyzing programs. The tool, which is used by both industry and academia, has already been successfully applied to uncover some bugs in real life C programs. It is quite efficient but is less ambitious than the *SG* and the *SG+R* algorithms on heap allocated data structures. In particular, it requires the user to supply annotations. It simplifies the analysis of loops by making it non conservative. Similar comments apply for the Prefix tool [6].

Compaq's Systems Research Center also aims at developing tools for static cleanness checking [9]. They have developed a tool called *Extended Static Checker* that uses verification technology. It requires simple annotations of pre- and post-conditions from the programmer and loop invariants. The tool uses program verification to find violations of certain cleanness conditions, such as NULL dereference and array bound violations. The tool was originally designed for Modula3 and was ported for Java. The *SG+R* algorithm compares favorably with the ESC tool for programs manipulating linked list since we automatically infer strong invariants. However, we currently cannot precisely handle arrays.

The algorithm of [38] is able to verify not only that a program is clean but also to show that it is correct with respect to a given partial specification. However, it depends on user-provided loop invariants, it is applicable only to a subset of Pascal, and its complexity is non elementary, i.e. $2^{2^{2^{\cdot^{\cdot}}}}$. Our comparisons indicate that the *SG+R* algorithm is faster by a factor of 10 to 100 than their algorithm.

Fradet, Gaugne, and Le Métayer suggest a Hoare-like logic that defines programs that obey the no-NULL dereference cleanness condition in a subset of C [27]. Although they propose a conservative algorithm that automatically checks a subset of the logic, this method cannot be extended to automatically check many programs since it cannot always phrase the loop invariants.

In [61] Scholz, Blieberger and Fahringer describe a different approach for detecting memory leaks. Instead of applying abstract interpretation to the program source, symbolic equations are extracted; these equations are solved in order to identify potential memory leaks. When the equations cannot be solved a false alarm may be generated. For loop-free programs this algorithm is complete, i.e., it does not generate false alarms. In contrast, we eagerly apply abstraction in order to always guarantee termination of the algorithm on programs containing loops while preserving many interesting properties of the program. As a result, our approach sometimes generates false alarms even for loop-free programs. Due to the lack of empirical evaluation, it is hard to asses the precision of the above cited result on programs with loops.

It is possible to use slicing techniques [34, 36, 26] to locate cleanness violations. However, current-slicing techniques yield very conservative results in languages that support pointers and references (especially in the presence of recursive data structures). In other words, it seems that current slicing tools will yield an excessive number of false alarms.

Pointer Analysis

Many conservative algorithms were proposed to analyze the content of the heap (e.g., [39, 7, 66, 28, 59]). Each of these algorithms can be used to provide conservative cleanness with different levels of cost and effectiveness. Our work contributes to the understanding of the benefits, costs, and limitations of pointer analysis in general and shape analysis in particular.

1.3 Outline of this Thesis

The rest of this thesis is organized as follows: Chapter 2 presents preliminaries for the string analysis algorithms. The SAS algorithm is formalized in Chapter 3 and CSSV in Chapter 4. Chapter 5 briefly compares between the CSSV and SAS algorithms. The CLLV algorithm is presented in Chapter 6. The thesis is concluded in Chapter 7.

Chapter 2

String Analyses — Preliminaries

Detecting errors in C programs is a difficult task even at runtime. This drives from the fact that the C runtime environment does neither define nor check for many errors, such as string and memory related errors. Additional information and formalization is required to define and check for buffer overflow and other errors. In this chapter we formalize the following:

CoreC We define the core elements of the C programming language. The aim of the CoreC language is to simplify the task of defining semantics, transformations and abstract interpretations of programs.

Contracts We use a contract language to define the string-related behavior of a procedure. For every procedure, the string analysis algorithms allow the programmer to provide a contract including: (i) a precondition (ii) a postcondition and (iii) the potential side-effects of the procedure. Contracts may refer to normal C expressions (including pointers) and can also refer to properties as defined in Section 2.2.

The procedure contracts are used for interprocedural analysis. Thus, contracts are used for determining the effect of a procedure call. As opposed to alternative interprocedural program analysis techniques, this approach has important advantages:

- Each procedure can be analyzed separately, exactly once. This is true even for (mutually) recursive procedures for which the recursive call is handled by simulating its effect as specified by the contract.
- The tool is applicable even if the source code is not available in its entirety.
- Contracts offer user control in a way similar to "design by contract" [49]. In particular, it enables our algorithms to locate more effectively the actual source location in which the error occurs.
- Contracts can improve the precision of the analysis by providing information which can be hard to statically infer via an interprocedural analysis.
- By using the contracts to analyze procedure calls, our algorithms apply a rather precise intraprocedural algorithm to reduce false alarms.

Semantics we define a concrete semantics [51] for programs which also includes checking of cleanness conditions and the given contracts.

The CoreC language is described in Section 2.1. Section 2.2 presents the contract language. Section 2.3 formalizes a concrete semantics for C programs. The contract language and the semantics are the foundation of both the SAS and the CSSV algorithms. As we shall see in the next chapters, the correctness of our algorithms is with respect to the semantics and the contract language presented in this chapter.

2.1 CoreC

The *CoreC* langauge is designed to include constructs of C that cannot be reduced to other constructs. The main benefit from CoreC is that it is small. This simplifies the task of defining semantics and analyses for C. We can summarize CoreC as a subset of C with the following restrictions:

- The `goto` statement is the only control-flow statement. The `if` statement contains two `goto` statements for the `true` and `false` branchs;
- expressions are side-effect free and cannot be nested;
- all assignments are statements;
- declarations do not have initializations;
- address-of a formal variable is not allowed.

An algorithm for transforming C programs to CoreC is presented in [69]. Given a C program, it generates a CoreC program by adding new temporaries. The generated program is equivalent to the original program in the sense that for defined programs according to ASNI-C and a given order of expression evaluation and a given order of argument evaluation the output of the CoreC program is the same as the output of the original C program [1]. The essence of CoreC which is important for our semantics and analyses is listed in Figure 2.1. We assume some countable set of variables, types and labels are given. We denote by *primitive* all values of all primitive types of C. Constants, and operators will not be further defined and neither will the additional syntactic functions shown in Table 2.1. We assume that each statement is labelled.

Example 2.1.1 The function `RTC_Si_SkipLine` (`SkipLine` for short) from EADS Airbus is shown in Figure 2.2 (a). This procedure inserts `NbLine` newline characters starting at the location pointed-to by `*PtrEndText`, appends a null-termination character and sets `*PtrEndText` to point to the end of the string. Due to multi-level pointer indirections, destructive updates, and pointer arithmetic, it is rather challenging to verify the absence of errors in this function. CSSV is able to statically verify the absence of string errors in this function, without reporting any false alarm.

The toy `main` procedure, shown in Figure 2.2 (a), calls `SkipLine` to insert a newline character, reads input from the standard input, and concatenates an additional newline by calling `SkipLine` again. This procedure has an off-by-one error. In the case of a user input of length `SIZE-1`, `buf` is full and there is no space for the additional newline. CSSV detects this error in `main` without reporting any false alarm.

Figure 2.2 (b) shows SkipLine's CoreC version. All double pointer indirections are simplified into two sequential pointer indirections and the `for` loop is changed to loop with `if` and `goto` statements.

[1] Note that in ANSI-C the order of expression evaluation is not defined. Our implementation follows the order imposed by Microsoft Visual-C. We could have followed other orders, e.g., the one imposed by GCC.

$S \in$ **Stmt**	statements	$t \in$ **Type**	defined types
$D \in$ **Decl**	declarations	$p \in$ **Proc**	declared procedures
$A \in$ **Assg**	assignments	$f \in$ **Field**	fields of declared types
$E \in$ **Expr**	expressions	$v \in$ **Var**	declared variables
$N \in$ **Name**	simple expressions	$c \in$ **Const**	used constants
$R \in$ **Deref**	dereference expressions	$l \in$ **Lab**	defined labels
$B \in$ **BExpr**	boolean expressions	$op_r \in \mathrm{Op}_\mathbf{r}$	relational operators
		$op_b \in \mathrm{Op}_\mathbf{b}$	binary operators
		$op_u \in \mathrm{Op}_\mathbf{u}$	unary operators

(a)

S	::=	$\{\ D \bowtie\mathbf{;}\ S \bowtie\mathbf{;}\ \}$	block of declarations and statements
	|	$l : A$	assignments
	|	l :**if** B **goto** l_1 **else goto** l_2	conditional goto
	|	l :**goto** l	unconditional goto
	|	l :**return** E	return from procedure call
D	::=	$t\ v$	declaration of scalar variable
	|	$D\ \mathbf{[}c\mathbf{]}$	declaration of array
A	::=	$N = E$	simple assignment
	|	$N = p\ \mathbf{(}N \bowtie\mathbf{,)}$	procedure call
	|	$R = c$	destructive update assigning constant
	|	$R = N$	destructive update assigning simple expression
E	::=	c	constant
	|	N	simple expression
	|	R	dereference
	|	$\mathbf{(}\ t\ \mathbf{)}\ N$	cast
	|	$op_u\ N$	unary operation
	|	$N_1\ \ op_b\ N_2$	binary operation
	|	$\mathbf{\&}N$	address-of simple expression
	|	$\mathbf{\&}R$	address-of dereference expression
N	::=	v	scalar variable
	|	$N\mathbf{.}f$	field operator
R	::=	$\mathbf{*}N$	simple dereference
	|	$(R)\mathbf{.}f$	dereference with field operator
B	::=	$N_1\ \ op_r\ N_2$	boolean expression

(b)

Figure 2.1: The CoreC language: (a) syntactic notations and (b) syntax. The notation *term*⋈*delimiter* indicates a concatenation of zero or more *terms* with the specified delimiter separated between terms.

```
void SkipLine(int NbLine, char** PtrEndText)
{
  int indice = 0;

  for ( ; indice<NbLine; indice++)
  {
    **PtrEndText = '\n';
    (*PtrEndText)++;
  }
  **PtrEndText = '\0';
}

void main()
{
  char buf[SIZE];
  char *r, *s;

  r = buf;
  SkipLine(1,&r);
  fgets(r,SIZE-1,stdin);
  s = r + strlen(r);
  SkipLine(1,&s);
}
```

(a)

```
void SkipLine(int NbLine, char** PtrEndText)
{
       int indice;
       char *PtrEndLoc;

[1]    indice=0;
[2]    begin_loop:
[3]        if (indice>=NbLine)
           goto end_loop;
[4]        PtrEndLoc = *PtrEndText
[5]        *PtrEndLoc = '\n';
[6]        *PtrEndText = PtrEndLoc + 1;
[7]        indice = indice + 1;
[8]        goto begin_loop;
[9]    end_loop:
[10]       PtrEndLoc = *PtrEndText;
[11]       *PtrEndLoc = '\0';
}

void main()
{
       char buf[SIZE];
       char *r, *s;
       char **ptr_r, **ptr_s;
       int number_bytes, len;

[1]    r = buf;
[2]    ptr_r = &r;
[3]    SkipLine(1,ptr_r);
[4]    number_bytes = SIZE-1;
[5]    fgets(r,number_bytes,stdin);
[6]    len = strlen(r);
[7]    s = r + len;
[8]    ptr_s = &s;
[9]    SkipLine(1,ptr_s);
}
```

(b)

Figure 2.2: `SkipLine` — a string manipulation function from EADS Airbus: (a) original function with a toy `main` procedure (b) the CoreC version of the procedures.

Name	Description
type : $\mathbf{Expr} \to \mathbf{Type}$	The type of an expression
sizeof : $\mathbf{Type} \to \mathbf{N}$	The allocation size in bytes of a type
$\mathit{max\_val}$: $\mathbf{Type} \to \mathit{primitive}$	The maximum value of a type
$\mathit{min\_val}$: $\mathbf{Type} \to \mathit{primitive}$	The minimum value of type t
$\mathit{field\_pos}$: $\mathbf{Field} \to \mathbf{N}$	The displacement in bytes of a field in its structure
cast: $(\mathbf{Type} \times \mathit{primitive} \times \mathbf{Type}) \to \mathit{primitive}$	Convert primitive from first type to primitive of the second type
`sizeof_ptr`	The size in bytes of pointer types

Table 2.1: Additional functions over CoreC expressions.

Attribute	Type	Intended Meaning
E.base	$\mathit{type}(E)$	The base address of E
E.offset	`int`	The displacement in bytes of E from its base
E.alloc	`int`	The number of bytes allocated from E
E.string	`Boolean`	Does E point-to to a null-terminated string?
E.strlen	`int`	The length in bytes of the string pointed-to by E
overlap(E_1, E_2)	`int`	The displacement of E_2 from E_1, i.e., $E_2 - E_1$
is_overlap(E_1, E_2)	`Boolean`	Do E_1 and E_2 point-to locations with the same base

Table 2.2: Attributes in the contract language. E_i's are pointer expressions.

2.2 Contracts

Contracts are used to describe expected inputs, potential side-effects, and expected output of functions. In this thesis, we write contracts in the style of Larch [43]. The syntax is declarative but may include CoreC expressions. Contracts are specified in the header file. Every prototype declaration of a function p has the form:

$$\begin{array}{lll} t\ p(\cdots) & \textbf{requires} & E_{Pre} \\ & \textbf{modifies} & E_{M_1}, E_{M_2}, \ldots, E_{M_n} \\ & \textbf{ensures} & E_{Post}; \end{array}$$

defining the precondition required to hold whenever p is invoked, the side-effects of the function p, i.e., the objects that may be modified when p is invoked, and the postcondition that is guaranteed to hold on every exit of the procedure. E_{Pre}, E_{M_i}, and E_{Post} are side effect free CoreC expressions over global and formal variables of p. We allow *attributes* formally defined in Table 2.2 and graphically displayed in Figure 2.3. The `base` attribute is of pointer type. A designated variable `return_value` denotes the return value of p. The special syntax $\lceil E \rceil_{entry}$ denotes the value of E when p is invoked. Although not required, the contract mechanism enables specifying pointer values. In addition a shorthand expression `is_within_bounds(`E`)` allows to indicate that E points within the bounds of a buffer. As we shall see, we define the bound of a buffer holding a string as the position of the null-termination byte. For a non-string buffer, the bound is determined by the allocation size. We do not allow contracts on variable number of parameters.

Example 2.2.1 The contract of the `strcpy()` function is shown in Figure 2.4. The precondition demands that upon entry: `src` is a valid string, the length of `src` is strictly less than the allocation

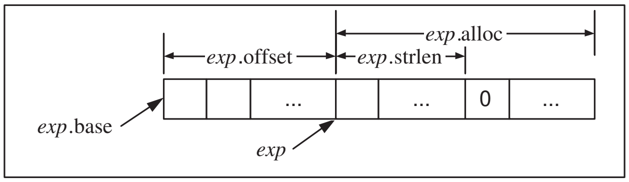

Figure 2.3: Graphical representation of contract-language attributes.

/* string.h */
char* strcpy(char *dst, const char *src)
requires src.string && dst.alloc > src.strlen &&
is_within_bounds(dst) && !is_overlap(dst,src)
modifies *dst
ensures dst.strlen == src.strlen &&
dst.alloc == $\lceil$dst.alloc$\rceil_{entry}$ &&
dst.string && `return_value`== dst ;

Figure 2.4: A contract for `strcpy()`.

size of `dst`, `dst` is within allocation bounds and that `src` and `dst` do not point to the same base address. `strcpy` may modify only the length and therefore the null termination of its `dst`-argument. In particular, it preserves the rest of `dst`'s attributes, e.g., `dst.alloc` is preserved. Finally, after `strcpy` returns, `dst` and `return_value` are updated appropriately.

Example 2.2.2 A contract for the `SkipLine` function is shown in Figure 2.5. The precondition demands that upon entry: `*PtrEndText` points to within the bounds of a buffer; the allocation space from the location `*PtrEndText` is greater than `NbLine`; and, `NbLine` is greater than zero. The function may modify the pointer `*PtrEndText` and the buffer pointed-to by `*PtrEndText`. The postcondition indicates that `*PtrEndText` points to a null-terminating string of length zero, and its value is advanced by `NbLine` bytes.

2.2.1 Contracts vs. Error Messages

There is a strong correlation between the contracts and the safety checks performed. For example, omitting `NbLine >= 0` from the precondition of `SkipLine` yields an error message during the analysis of `SkipLine`. The message indicates that the postcondition

$$\texttt{*PtrEndText.offset} == \lceil \texttt{*PtrEndText.offset} \rceil_{entry} + NbLine$$

may not hold. Interestingly, the counter-example produced by CSSV for this message shows that this postcondition does not hold when the value of `NbLine` is negative.

Providing a stronger precondition than the weakest precondition ([17]) can yield error messages on some procedure invocations. For example, requiring that `*PtrEndText` points-to a null-terminated

```
void SkipLine(int NbLine, char** PtrEndText)
  requires  is_within_bounds(*PtrEndText) &&
            *PtrEndText.alloc > NbLine && NbLine >= 0
  modifies  *PtrEndText
  ensures   *PtrEndText.string &&
            *PtrEndText.strlen == 0 &&
            *PtrEndText.alloc = ⌈*PtrEndText.alloc⌉_entry + NbLine &&
            *PtrEndText.offset == ⌈*PtrEndText.offset⌉_entry + NbLine ;
```

Figure 2.5: A contract for `SkipLine`.

string will result in a report on a potential violation of the precondition of `SkipLine` at the call in line [9] of `main`.

2.3 A Non-Standard Operational Semantics

The C programming language does not rigorously define semantics of programs. In the ANSI-C standard there is an informal notion of defined and undefined behaviors. However, the exact behavior can change, and often does, from one implementation of a compiler to another. Due to the following features of the language, it is not trivial to define a semantics for C:

Address-of operation enables changing the value of a variable without assigning to the variable. It also permits pointers to invisible variables.

Allocation library functions (e.g., `malloc`) provide an unformatted contiguous memory locations, while from the logic point of view there is a “hierarchy” of objects where one object may contain objects of different types. Moreover, objects are type-less, thus providing flexibility, and allowing accesses to a location according to different types. Therefore, it is difficult to define and check the legitimacy of an access.

Pointer arithmetic is frequently used and has a defined result. However, checking its validity is impossible without additional information. In particular, is it valid for pointer arithmetic to result in the first location beyond the high end of a buffer. However, this location can be the first location of a different allocated buffer which is valid to access and dereference.

Cast operation exposes the internal memory layout e.g., by allowing casting from integer to pointer type.

This section defines an instrumented operational semantics for C that dynamically verifies the absence of certain safety and cleanness violations, such as out-of-bounds access, uninitialized variables, invalid casting, illegal references and dereferences, usage of dead storage. In addition, the semantics verifies the correctness of the contract. This is rather straightforward for the precondition and postcondition, but more subtle for the `modifies` clause. The semantics verifies that each assigned location is either stated as modified or not reachable from previous activation records. The semantics forbids programs with undefined ANSI-C behavior but it also checks additional requirements reflecting good programming styles such as references beyond the null-termination byte. This semantics provides the foundation for our static checking tools. The soundness of the tools are proven with respect to this semantics.

The semantics handles the above listed complications of C by:

- An environment which explicitly stores the static, stack or global location of each variable and treats this location as any other location.
- Storing the size of memory contents.
- Defining and verifying validity of every expression, in particular pointer arithmetic and casting, by using the additional information.
- Allocation of one extra byte for each buffer to fully allow pointer arithmetic but disallow pointer arithmetic resulting in a different buffer.
- Modelling the entire stack and small step execution of procedure calls.

2.3.1 Overview of the Semantics

There are existing suggestions for C semantics for the purpose of verification, e.g., [2, 31, 53]. The goal of our non-standard semantics is to provide a full, clear and simple definition for C programs that includes verification of cleanness conditions. Since our aim is to abstract this semantics, there is no concern regarding the efficiency of the semantics. Instead we define semantics that can be rather precisely abstracted.

The general idea is to explicitly represent properties of *locations*. A location is the most basic memory unit, thus a byte in most operating systems. The semantics maintains the *base address* of every memory location, i.e., to which *buffer* does a location belong. In addition, the allocation size of each base address is maintained. This extra information provides the ability to validate that expressions have a defined behavior. For example, the semantics disallows pointer arithmetic between different base addresses. Instead of storing the content of each location separately, the semantics maps locations to a pair:(i) the logical content and, (ii) the size (number of locations) of this content. Thus, an integer value, say 10, stored at 4 locations l_1, l_2, l_3, l_4 is modelled by a content mapping from l_1 to 10 and a size mapping l_1 to 4.

2.3.2 Concrete State

We define a *concrete state* that allows us to reason about intermediate stages during the execution of a program. A *state* consists of two parts. The first part, called *store*, reasons about the values of allocated locations (addresses). The second part, called *stack*, maps variables to their stack location and maintains an activation record. In order to model statements which manipulate variable addresses, e.g, $*x$, our semantics represents every stack as a mapping from variables into addresses.

Definition 2.3.1 A **concrete state at a procedure** P is a pair $\sigma^\natural = (\mathit{store}^\natural, \mathit{stack}^\natural)$ where:

- $\mathit{store}^\natural$ is a tuple $(\mathcal{L}^\natural, \mathcal{BA}^\natural, \mathit{size}^\natural, \mathit{content}^\natural, \mathit{numBytes}^\natural, \mathit{base}^\natural, \mathit{allocatedAt}^\natural)$ where:
  - $\mathcal{L}^\natural \subseteq \mathbf{N}$ is a finite set of all static, stack, and dynamically allocated locations.
  - $\mathcal{BA}^\natural \subseteq \mathcal{L}^\natural$ is the set of base addresses in $\mathcal{L}^\natural$.
  - $\mathit{size}^\natural : \mathcal{BA}^\natural \to \mathbf{N}$ is the number of allocated bytes of a base address.

  - *content*$^\natural$: $\mathcal{L}^\natural \rightarrow$ *val* defines the memory content where

$$val = \{\textit{uninit}, \textit{undef}\} \cup primitive \cup \mathcal{L}^\natural$$

    is the set of possible values.

    * The value *uninit* represents uninitialized values
    * *undef* represents results from illegal memory accesses
    * *primitive* refers to the set of C primitive type values

  - *numBytes*$^\natural$: $\mathcal{L}^\natural \rightarrow \mathbf{N}$ indicate the number of bytes of the value stored in a given location.
  - $base^\natural$: $\mathcal{L}^\natural \rightarrow \mathcal{BA}^\natural$ maps a location $l^\natural$ to the base address it is allocated with.
  - *allocatedAt*$^\natural$: $\mathcal{BA}^\natural \rightarrow$ Lab defines for each base address at which label (statement) was it allocated.

- *stack*$^\natural$ is a tuple $(\textit{depth}^\natural, \textit{loc}^\natural)$ where:

  - *depth*$^\natural \in \mathbf{N}$ indicates the current depth of activation records. Depth zero is the first entry in the program and each call increases the depth by one and each return decreases by one.
  - $loc^\natural$: (Var $\times$ $\mathbf{N}$) $\hookrightarrow$ $BA^\natural$ maps variables and depths into their assigned global or stack location (which is always a base address). There is a *loc*$^\natural$ mapping for each activation record. This mapping is defined only for variables visible at the given activation record depth.

We shall first establish intended state's properties:

$\mathcal{L}^\natural$: The set of locations is separated to groups of *contiguous* allocated regions. These sets are disjoint. In addition we define an ordering $<$ on $\mathcal{L}^\natural$ which defines the *least* location of a set of locations.

$\mathcal{BA}^\natural$: A *base address* is the least address of an independent contiguous allocated memory region. Specifically, every primitive type object has a base address, an array has one base address, and every structure has one base address. A *buffer* is a set of locations with the same base address.

$base^\natural$: Every allocated location is part of a contiguous allocated region. This mapping maps a location to the base address of the contiguous region to which it belongs.

size$^\natural$: A base address is associated with a size. As we shall see, this mapping, together with $base^\natural$, enables the verification of pointer arithmetic.

content$^\natural$: This mapping defines the content of each location. The number of bytes of a content is stored in *numBytes*$^\natural$. For example, a location $l^\natural$ which stores an integer value of 4 bytes has a mapping *numBytes*$^\natural(l^\natural) = 4$. We propose this representation, which is somewhat less natural than representing the content of each byte separately, since it provides a logical view of the store. We assume that each primitive value is associated with a type. For example, there is a primitive value 10 of type integer and a primitive value 10 of type short.

depth$^\natural$: The first state of a program contains a depth zero. The execution of a call statement increases the depth by one and each return decreases by one.

$loc^\natural$: This mapping maps a pair of variable and nesting depth into a location. This enables to store different activation records of the same procedure.

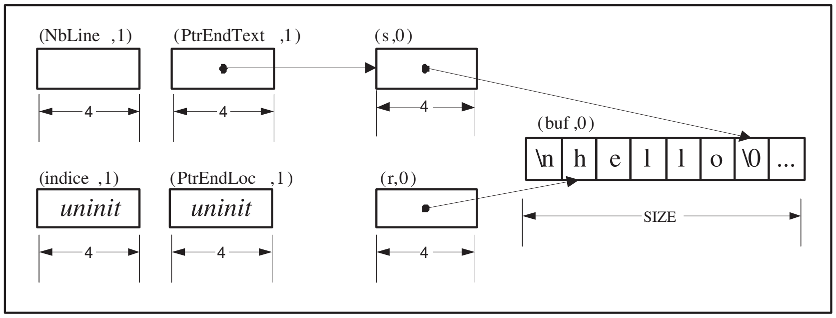

Figure 2.6: A concrete state arising at entry to `SkipLine` invoked by the second call from `main`. For clarity, the allocation size of `buf` is only shown symbolically.

For each property we define a corresponding function for extracting this property of a state. For example, $loc^\natural(\sigma^\natural)$ returns the $loc^\natural$ part of $\sigma^\natural$. Hence, $loc^\natural(\sigma^\natural)(v, depth^\natural)$ returns the $loc^\natural$ mapping of the pair $(v, depth^\natural)$ in the state $\sigma^\natural$.

Example 2.3.2 A concrete state that arises at entry to `SkipLine` when invoked by the second call in `main` is shown in Figure 2.6. Each box represents an allocated contiguous memory region starting at a base-address. We display their allocation sizes underneath the boxes. Here, we assume that integers and pointers are four bytes long and a character is one byte long. We draw a pair (v, i) above a box whose base address is the location of variable v in depth i, i.e., $loc^\natural(v, i)$. The invocation of main is at depth zero and the invocation of `SkipLine` is at depth one. The value inside each box shows the corresponding store content. Pointer values are drawn as edges.

2.3.3 Meaning of Expression

Given a state, there are two meanings for an expression: *L-value* and *R-value*. An L-value represents a memory location value. It is called left value, since it is the value of the left hand side of an assignment statement. An R-value (right value) is the value of an expression. The notation $lv(E, \sigma^\natural)$ ($rv(E, \sigma^\natural)$) is used for the L-value (R-value) of an expression E at a state $\sigma^\natural$. We omit $\sigma^\natural$ when it is clear from the context.

L-value

Table 2.3 shows the L-value of expressions. The L-value of a scalar variable is the static or stack location of the variable at the current depth. The depth is known as part of the state. The L-value of a dereference is the content of the location dereferenced. For field operator, the location is computed using the displacement of the field within the structure.

R-value

Table 2.4 defines the R-value of expressions. We assume that the R-value of a literal c, denoted as $[\![c]\!]$, is given. Similarly, operator meanings are given by corresponding semantic functions. For example, the R-value of a binary operation *op* on actual arguments values v_1 and v_2 with types t_1 and t_2 respectively is denoted by $[\![\, op, t_1, t_2]\!](v_1, v_2)$. The R-value of complex operations is inductively defined according to the R-value of the operands. When an address is used in an expression, the R-value is obtained using

Expression		L-Value
v	scalar variable	$loc^\natural(\sigma^\natural)(v, depth^\natural(\sigma^\natural))$
$*N$	simple dereference	$content^\natural(\sigma^\natural)(lv(N))$
$E.f$	all field operator expressions	$lv(E) + field\_pos(f)$

Table 2.3: L-value of expressions in a concrete state $\sigma^\natural$

Expression		R-Value
c	constant	$[\![c]\!]$
$N_1\ op_r\ N_2$	boolean expression	$[\![\ op_r, type(N_1), type(N_2)]\!](rv(N_1), rv(N_2))$
$op_u\ N$	unary operation	$[\![\ op_u, type(N)]\!](rv(N))$
$N_1\ op_b\ N_2$	binary operation	$[\![\ op_b, type(N_1), type(N_2)]\!](rv(N_1), rv(N_2))$
$\&E$	address-of	$lv(E)$
$(\ t\ )\ N$	cast	$cast(type(N), rv(N), t)$
E	any other expression	$\begin{cases} content^\natural(\sigma^\natural)(lv(E)) & sizeof(type(E)) = numBytes^\natural(\sigma^\natural)(lv(E)) \\ undef & otherwise \end{cases}$

Table 2.4: R-value of expression of in a concrete state $\sigma^\natural$.

the corresponding L-value, e.g., in the case of an address-of expression. In the general case (last row in Table 2.4), which includes dereferences and simple expressions the R-value of an expression E is the content of the L-value of E.

The contract language contains attributes to describe important properties regarding the string manipulated by the procedure. Table 2.5 defines instrumented information which is used for computing the meaning of these attributes. Since contracts are side-effect free these attributes do not have an L-value. Table 2.6 lists the R-value of the attributes. Some are computed using the *base* and *size* emphasizing the need for these mappings in the state representation.

2.3.4 Semantics

For every statement and expression, E, there is a precondition that must hold on the state in order for E to have a clean and defined behavior, and thus can be executed safely. For some expressions the precondition is `true`, thus it always has a defined behavior. Similarly, some statements change the state while others do not. This section defines the precondition and transition on a state of each expression and statement. The control-flow and execution of statements is in the regular sequential manner. Briefly, the meaning of the condition of an `if` statement is interpreted to either the *true* or *false* value. According to the value, the control flows to either the *true* or *false* branch. We omit formal definition of the control flow of the execution of a program.

Notation

The notation $t[x \mapsto y]$ denotes a mapping in which x is mapped to y and every other element is mapped according to t. We also extend this notation to sets allowing simulation assignment. Thus, we write $t\{x_i \mapsto e_i | \rho\}$ to express a new mapping in which each of the x_i is mapped to e_i for x_i's satisfying ρ.

Shorthand	Meaning
$\mathit{offset}^\natural(\sigma^\natural)(l^\natural)$	$l^\natural - \mathit{base}^\natural(\sigma^\natural)(l^\natural)$
$\mathit{alloc}^\natural(\sigma^\natural)(l^\natural)$	$\mathit{size}^\natural(\sigma^\natural)(\mathit{base}^\natural(\sigma^\natural)(l^\natural)) - \mathit{offset}^\natural(\sigma^\natural)(l^\natural)$
$\mathit{strlen}^\natural(\sigma^\natural)(l^\natural)$	$\begin{cases} i & \mathit{content}^\natural(\sigma^\natural)(l^\natural + i) = \text{'}\backslash 0\text{'} \wedge 0 \leq i < \mathit{alloc}^\natural(\sigma^\natural)(l^\natural) \wedge \\ & \neg\exists j : 0 \leq j < i \wedge \mathit{content}^\natural(\sigma^\natural)(l^\natural + j) = \text{'}\backslash 0\text{'} \\ \mathit{undef} & \text{otherwise} \end{cases}$
$\mathit{string}^\natural(\sigma^\natural)(l^\natural)$	$\begin{cases} \mathit{true} & \mathit{strlen}^\natural(\sigma^\natural)(l^\natural) \neq \mathit{undef} \\ \mathit{false} & \text{otherwise} \end{cases}$
$\mathit{is\_overlap}^\natural(\sigma^\natural)(l_1^\natural, l_2^\natural)$	$\begin{cases} \mathit{true} & \mathit{base}^\natural(\sigma^\natural)(l_1^\natural) = \mathit{base}^\natural(\sigma^\natural)(l_2^\natural) \\ \mathit{false} & \text{otherwise} \end{cases}$
$\mathit{overlap}^\natural(\sigma^\natural)(l_1^\natural, l_2^\natural)$	$\begin{cases} \mathit{offset}^\natural(\sigma^\natural)(l_2^\natural) - \mathit{offset}^\natural(\sigma^\natural)(l_1^\natural) & \mathit{is\_overlap}^\natural(\sigma^\natural)(l_1^\natural, l_2^\natural) \\ \mathit{undef} & \text{otherwise} \end{cases}$

Table 2.5: Instrumented information for a concrete state $\sigma^\natural$.

Attribute	R-value
E.base	$\mathit{base}^\natural(\sigma^\natural)(rv(E))$
E.offset	$\mathit{offset}^\natural(\sigma^\natural)(rv(E))$
E.alloc	$\mathit{alloc}^\natural(\sigma^\natural)(rv(E))$
E.strlen	$\mathit{strlen}^\natural(\sigma^\natural)(rv(E))$
E.string	$\mathit{string}^\natural(\sigma^\natural)(rv(E))$

Table 2.6: R-value of attributes in a concrete state $\sigma^\natural$.

We use $e[e'/x]$ to denote the result of substituting e' for x in e. Similarly, $e\{e'/x|\rho\}$ expresses a set of substitutions. Finally, $[\![S]\!]\sigma^\natural$ denotes the state obtained by applying statement S to state $\sigma^\natural$.

Declaration

The first entry in Table 2.7 shows the semantics of declarations. The precondition verifies the existence of the required contiguous memory locations. Both the store and the stack part of the state are changed. The store is changed to reflect the newly allocated region size and allocation label. The stack is changed to contain an additional mapping $loc^\natural$ of the declared variable to its allocated storage. The allocated region contains one extra location for the high end location. This extra location is set as part of the allocated region to allow reference. As we shall see, the semantics of expression verifies that this location is not dereferenced.

Expression

One of the important and convenient features of CoreC is that expressions are side-effect free. Therefore they have preconditions but do not alter the state. Table 2.8 shows the precondition of each expression type. The precondition of an expression is defined by recursively traversing the syntax of the expression. For example, on a dereference `*p` the reference to `p` is first verified. We check for usage of uninitialized variables and a reference to an undefined location. On a dereference, in addition to checking reference, the reference should be within the bounds of an allocated buffer (and not at the high end). We verify that a cast expression to type t is valid if it is within the defined values of type t.

Pointer arithmetic is the most complex expression to verify. The result of pointer arithmetic, `p + i` depends on the type of the pointer which can defer for the allocation type of the buffer pointed-to by the pointer. The resultant address, computed by `p + i*sizeof(`*type*`(*p))`, can certainly be allocated and valid to reference, yet outside the boundaries of the buffer pointed to by `p`. We require that the R-value be within the bounds of the contiguous memory region.

Allocation

Although memory allocation and deallocation are function calls, they have special meaning. Therefore they are shown as statements in Table 2.7. An allocation is valid if the allocation size is greater than zero and if the required contiguous locations are available.

When allocating memory, an extra byte is allocated to enable detection of pointer arithmetic errors. The size of the allocated region is according to the actual parameter. The precondition of a deallocating expression is that the argument points to an allocated base address. This base address is set to have a zero size. The deallocated locations are not removed from $\mathcal{L}^\natural$ which facilitate the detection of future access to this dead storage.

Assignment

The left-hand side and the right hand side are verified by structural induction. In addition, the semantics demands that only one null-termination byte exists in a buffer. The number of changed bytes is determined by the size of the argument's type. In CoreC assignments are typed, i.e., the type of the left and right hand sides are the same. An assignment changes the content of the L-value of the left-hand side expression. However, it can also change mappings of other locations as well. For example, consider the code in Figure 2.7. At line [1], four bytes starting at the location of `i` are written. At line [4] and [5] only one of `i`'s locations is overwritten, changing the R-value of `i`.

When updating a store, only *admissible* states are created.

Statement	Precondition	Transition
l: declaration of v of type t	there exists $l_0^\natural$ such that: $\{l_0^\natural + i \mid 0 \le i \le \mathit{sizeof}(t)\} \cap \mathcal{L}^\natural = \emptyset$	let $\mathit{store}'^\natural = (\mathcal{L}^\natural \cup \{l_0^\natural + i \mid 0 \le i \le \mathit{sizeof}(t)\},$ $\mathcal{BA}^\natural \cup \{l_0^\natural\},$ $\mathit{size}^\natural[l_0^\natural \mapsto \mathit{sizeof}(t)],$ $\mathit{content}^\natural\{l_0^\natural + i \mapsto \mathit{uninit} \mid 0 \le i \le \mathit{sizeof}(t)\},$ $\mathit{numBytes}^\natural\{l_0^\natural + i \mapsto 0 \mid 0 \le i \le \mathit{sizeof}(t)\},$ $\mathit{base}^\natural\{l_0^\natural + i \mapsto l_0^\natural \mid 0 \le i \le \mathit{sizeof}(t)\},$ $\mathit{allocatedAt}^\natural[l_0^\natural \mapsto l])$ $\mathit{stack}'^\natural = (\mathit{depth}^\natural, \mathit{loc}^\natural[(v, \mathit{depth}^\natural) \mapsto l_0^\natural])$ in $(\mathit{store}'^\natural, \mathit{stack}'^\natural)$
l: $N = (*t)$malloc(E) dynamic allocation	there exists $l_0^\natural$ such that: $\{l_0^\natural + i \mid 0 \le i \le \mathit{rv}(E)\} \cap \mathcal{L}^\natural = \emptyset$ and $\mathit{rv}(E) > 0$	let $\mathit{store}'^\natural = (\mathcal{L}^\natural \cup \{l_0^\natural + i \mid 0 \le i \le \mathit{rv}(E)\},$ $\mathcal{BA}^\natural \cup \{l_0^\natural\},$ $\mathit{size}^\natural[l_0^\natural \mapsto \mathit{rv}(E)],$ $\mathit{content}^\natural\{l_0^\natural + i \mapsto \mathit{uninit} \mid 0 \le i \le \mathit{rv}(E)\},$ $\mathit{numBytes}^\natural\{l_0^\natural + i \mapsto 0 \mid 0 \le i \le \mathit{rv}(E)\},$ $\mathit{base}^\natural\{l_0^\natural + i \mapsto l_0^\natural \mid 0 \le i \le \mathit{rv}(E)\},$ $\mathit{allocatedAt}^\natural[l_0^\natural \mapsto l])$ in $[\![N = l_0^\natural]\!](\mathit{store}'^\natural, \mathit{stack}^\natural)$
$E_1 = E_2$ all assignments	$\mathit{rv}(E_2) \neq$ '\0' $\vee \neg \mathit{string}^\natural(\sigma^\natural)(\mathit{lv}(E_1)) \wedge$ $\mathit{alloc}^\natural(\sigma^\natural)(\mathit{lv}(E_1)) \ge \mathit{sizeof}(\mathit{type}(E_2))$	$(\, (\, \mathcal{L}^\natural, \mathcal{BA}^\natural, \mathit{size}^\natural,$ $\mathit{content}^\natural[\mathit{lv}(E_1) \mapsto \mathit{rv}(E_2)]$ $\{\mathit{lv}(E_1) + i \mapsto \mathit{undef} \mid \ 1 \le i < \mathit{sizeof}(\mathit{type}(E_2))\}$ $\{l'^\natural \mapsto \mathit{undef} \mid l'^\natural < \mathit{lv}(E_1) < l'^\natural + \mathit{numBytes}^\natural(l'^\natural)\},$ $\mathit{numBytes}^\natural\{\mathit{lv}(E_1) \mapsto \mathit{sizeof}(\mathit{type}(E_2))\},$ $\mathit{base}^\natural, \mathit{allocatedAt}^\natural),$ $\mathit{stack}^\natural)$
N = free(E) deallocation	$\mathit{rv}(E) \in \mathcal{BA}^\natural \ \wedge \ \mathit{size}^\natural(\sigma^\natural)(\mathit{rv}(E)) > 0$	$(\, (\mathcal{L}^\natural, \mathcal{BA}^\natural, \mathit{size}^\natural[\mathit{rv}(E) \mapsto 0\,], \mathit{content}^\natural, \mathit{base}^\natural, \mathit{allocatedAt}^\natural),$ $\mathit{stack}^\natural)$

Table 2.7: Semantics of statements in $\sigma^\natural = (\mathcal{L}^\natural, \mathcal{BA}^\natural, \mathit{size}^\natural, \mathit{content}^\natural, \mathit{base}^\natural, \mathit{allocatedAt}^\natural), \mathit{stack}^\natural)$.

Expression	Precondition
E R-value meaning	$rv(E) \neq uninit \ \wedge \ rv(E) \neq undef \wedge$ $min\_val(t) \leq rv(E) \leq max\_val(t)$
E L-value meaning reference	$lv(E) \neq undef \ \wedge \ alloc^\natural(\sigma^\natural)(lv(E)) > 0 \ \wedge$ $(\neg string^\natural(\sigma^\natural)(rv(N.base)) \vee strlen^\natural(\sigma^\natural)(rv(N)) \geq 0)$
$*N$ dereference	$(string^\natural(\sigma^\natural)(rv(N.base)) \ \wedge strlen^\natural(\sigma^\natural)(rv(N)) \geq 0 \ \vee$ $(\neg string^\natural(\sigma^\natural)(rv(N.base)) \ \wedge \ alloc^\natural(\sigma^\natural)(rv(N)) > 0))$

$(\ t\ )\ N$
cast
	let
	  val = $cast(type(N), rv(N), t)$
	in
	  $min\_val(t) \leq$ val $\leq max\_val(t)$

$N_1 \ op_b \ N_2$
where:
$type(N_1) = t*$
$type(N_2) =$ `int`
	let
	  targetOffset = $offset^\natural(\sigma^\natural)(rv(N_1)) \ [\![\ op_b]\!] (rv(N_2) * sizeof(t))$
	in
	  $0 \leq$ targetOffset $\leq size^\natural(\sigma^\natural)(base^\natural(\sigma^\natural)(rv(N_1)))$

N_1/N_2 division	$rv(N_2) \neq 0$

Table 2.8: Precondition of expressions on $\sigma^\natural$.

Definition 2.3.3 A concrete state $\sigma^\natural$ is **admissible** if for every location $l^\natural \in \mathcal{L}^\natural(\sigma^\natural)$ such that

$$content^\natural(\sigma^\natural)(l^\natural) \neq undef$$

then

$$\begin{gathered} content^\natural(\sigma^\natural)(l^\natural + i) = undef, \ 1 \leq i < numBytes^\natural(\sigma^\natural)(l^\natural) \ \wedge \\ \forall l'^\natural \in \mathcal{L}^\natural(\sigma^\natural) : (content^\natural(\sigma^\natural)(l'^\natural) = undef \ \vee \ \neg(l'^\natural < l^\natural < l'^\natural + numBytes^\natural(\sigma^\natural)(l'^\natural))) \end{gathered}$$

Thus an admissible state does not contain any "overlapping" contents. Our semantics only yields admissible states. This is achieved by defining the semantics of assignments to set all mapping that are overwritten to *undef*. We denote by $\Sigma^\natural(P)$ the set of all admissible states arising during executions of program fragment P.

The semantics of assignments generates admissible states. This is achieved by setting the content of all locations that are written indirectly to *undef*. In addition, all values that might have been partly overwritten are also set to *undef*. Figure 2.7 illustrates the generated undefined values due to assignments. For clarity reasons, it presents the R-value of the interesting expressions.

Procedure Call

The small-step operational semantics of a procedure call consists of an *entry* step, an execution of the statements of the callee, and an *exit* step. Table 2.9 defines the semantics of the entry and exit steps for the statement:

$$N = g(E_1, E_2, \cdots E_n)$$

```
main() {
   int i;
   char *p;
   char *q;

[1]    i = 5;
                          rv(i) == 5
[2]    p = &i;
                          rv(i) == 5
                          rv(*p) == undef
[3]    q = p + 2;
                          rv(i) == 5
                          rv(*p) == undef
                          rv(*q) == undef
[4]    *q = 'y';          // i's third byte is overwritten
                          rv(i) == undef
                          rv(*p) == undef
                          rv(*q) == 'y'
[5]    *p = 'n';          // i's first byte is overwritten, third byte is left as-is
                          rv(i) == undef
                          rv(*p) == 'n'
                          rv(*q) == 'y'
}
```

Figure 2.7: Creation of undefined values as a result of different update statements.

where g is declared as:

$$\begin{array}{lll} t\ g(f_1, f_2, \cdots f_n) & \textbf{requires} & E_{Pre} \\ & \textbf{modifies} & E_{M_1}, E_{M_2}, \ldots, E_{M_m} \\ & \textbf{ensures} & E_{Post}; \end{array}$$

and the return statement is

$$\text{return } E_r$$

On the entry the precondition of the callee is verified. This is done according to the precondition defined in g's contract and substituting the formals with the actuals. The state at the entry to the procedure is computed by: first a new depth is created. Next, each formal variable's declaration is processed and assigned the actual parameter value.

A `return` statement is valid if g's postcondition holds. The ensures expression of a contract may include pre expressions, $\lceil E \rceil_{entry}$, denoting the value of E at the entry to the procedure. In order to verify the postcondition, the state at the entry to the procedure is used to compute the R-value of the pre-expressions. We generate the expression E_p by replacing in g's postcondition formals with actuals, pre-expressions with their meaning and also the designated `return_value` variable with the actual return value. Next, the expression is verified to hold. In addition, the precondition of a `return` verifies that the return-value matches the return-type.

In the contract of a function, the locations that may be modified during the procedure invocation are indicated. To verify that every location modified during the execution of the procedure is indicated in the `modifies` clause, the state at the entry to the procedure is used to compute the set of *modified locations*.

Definition 2.3.4 A **location $l^\natural$ is reachable from expression** E if the L-value of E can (indirectly) include $l^\natural$ (i.e., there is an expression starting with E whose L-value is $l^\natural$). The function *reach*$(E, \sigma^\natural)$ returns all reachable locations from the expression E at state $\sigma^\natural$.

Given the mod expression $E_{M_1}, E_{M_2}, \cdots E_{M_n}$ stated in the contract of function g, the set of modified locations for state $\sigma^\natural$ is defined by

$$mod_g(\sigma^\natural) \stackrel{\text{def}}{=} \{reach(E_{M_i}, \sigma^\natural) | 1 \leq i \leq n\}$$

On assignments the semantics verifies that contents of locations which are required to be preserved according to the contract are not changed. We require that every location that is modified during the execution and reachable from an external variable is specified in the `modifies` clause. Thus, a procedure may change a local variable not indicated in the contract as modified. This is done as follows:

- The mod expressions specified in the contract are according to the values at the entry state, $\sigma^\natural_e$. We compute the set $mod_g(\sigma^\natural_e)$ — the possibly modified locations.
- At an assignment $E_1 := E_2$ in $\sigma^\natural$ of procedure g we require that:

$$\begin{array}{l} lv(E_1, \sigma^\natural) \in mod_g(\sigma^\natural_e) \vee \\ \forall v \in \textbf{Var}, \forall i \leq depth^\natural(\sigma^\natural_e) : lv(E_1, \sigma^\natural) \notin reach(loc^\natural(\sigma^\natural)(v, i), \sigma^\natural) \end{array}$$

An exit step changes the state reflect that all local variables are removed from the stack. The depth of the current activation record is decreased, and the left-hand side expression of the call statement is assigned the return value. At the end-of-scope of a variable, its stack location (and all inner locations) are deallocated similarly to a `free` statement. In addition, the $loc^\natural$ mapping is updated to reflect that the variable is no longer allocated.

Step	Precondition	Transition
entry	$rv(E_{Pre}\{E_i/f_i \mid 1 \le i \le n\})$	let $\sigma_0^\natural = (store^\natural \sigma^\natural), (depth^\natural(\sigma^\natural) + 1, loc^\natural(\sigma^\natural)))$ $\sigma_i^\natural = [\![f_i = E_i]\!]([\![t_i\ f_i]\!](\sigma_{i-1}^\natural))$ where $1 \le i \le n$ in $\sigma_n^\natural$
exit	let $E_p = E_{Post}\ \{E_i/f_i \mid 1 \le i \le n\}$ $[rv(E_r)/\texttt{return\_value}]$ $\{rv(e, \sigma_e^\natural)/\lceil e \rceil_{entry} \mid \lceil e \rceil_{entry} \in E_{Post}\}$ in $rv(E_p, \sigma^\natural) \wedge$ $min\_val(t) \le rv(E, \sigma^\natural) \le max\_val(t)$	let $\sigma_0^\natural = \sigma^\natural$ $\texttt{temp} = rv(E_r, \sigma^\natural)$ $\sigma_i^\natural = [\![$end of scope of $v_i\ ]\!]\sigma_{i-1}^\natural$ for all local variables of g: $v_1, v_2, \cdots, v_k$ $\sigma_{k+1}^\natural = (store^\natural, (depth^\natural(\sigma_k^\natural) - 1, loc^\natural(\sigma_k^\natural)))$ in $[\![N = \texttt{temp}]\!]\sigma_{k+1}^\natural$
end of scope of v		$((\ \mathcal{L}^\natural(\sigma^\natural), \mathcal{BA}^\natural(\sigma^\natural),$ $size^\natural(\sigma^\natural)[loc^\natural(\sigma^\natural)(v, depth^\natural(\sigma^\natural)) \mapsto 0]$ $content^\natural(\sigma^\natural), base^\natural(\sigma^\natural)),$ $(depth^\natural(\sigma^\natural), loc^\natural(\sigma^\natural \setminus v))$

Table 2.9: Semantics of procedure call in a concrete state $\sigma_e^\natural$ arises at the entry to g and concrete state $\sigma^\natural$ arises at the exit state.

2.4 Simulation

The *simulation* property is used to describe that a transformation preserves certain properties. A simulation can be defined among programs of different languages and may simulate only a subset of values and properties of the original program. A simulation can be defined for whole programs or for smaller fragments, such as procedures. $Tr(P)$ denotes the transformation of fragment P. The transformation of statement S, $Tr(S)$, may include more than one statement. We denote by $\Sigma^\natural(P)$ ($\Sigma^\natural(Tr(P))$ the set of states arising during the executions of P ($Tr(P)$), respectively. We formally define simulation as follows:

Definition 2.4.1 A transformation is a *simulation* of program fragment P with respect to simulation relation

$$\varphi \subseteq \Sigma^\natural(P) \times \Sigma^\natural(Tr(P))$$

if the following two requirements hold:

Entry Requirement For the entry state $\sigma^\natural \in \Sigma^\natural(P)$ exists an entry state $\sigma^\natural_{tr} \in \Sigma^\natural(Tr(P))$ such that:

$$(\sigma^\natural, \sigma^\natural_{tr}) \in \varphi$$

Induction Requirement Every step preserves the simulation: If $(\sigma^\natural, \sigma^\natural_{tr}) \in \varphi$ and $\sigma'^\natural$ is derived by executing statement S on $\sigma^\natural$ then there exists a $\sigma'^\natural_{tr} \in [\![Tr(S)]\!](\sigma^\natural_{tr})$ such that:

$$(\sigma'^\natural, \sigma'^\natural_{tr}) \in \varphi$$

As we shall see, both the SAS and CSSV algorithms defines transformations which simulate properties of the original program. The simulation property is used to describe the correctness of the transformation which is the first step in proving the soundness of the algorithm.

2.5 Conclusion

In this chapter we defined important foundations used in our string analysis algorithms, but which are useful for other analysis and programming tools. The CoreC language is important for the definition of a clean and simple semantics and for a fully defined static analysis. The contract language, a key element in the effectiveness and efficiency of the algorithms, is designed not just for expressiveness but also for generating analyzable contracts. The defined semantics is designed to be full but simple, without any redundant information. The added preconditions provide safety checks which determines clean expressions. The goal of the semantics is not efficiency. On the contrary, for example, we assume unlimited space. The motivation of many of the features of the semantics is, as we shall see in the next chapters, the ability to define in a similar format an abstract semantics that can statically determine potential violations of preconditions. Finally, the definition of simulation enables the discussion of the correctness and soundness of transformations which are used in both the SAS and CSSV's algorithms.

Chapter 3

SAS: String Analysis for Sub C

This chapter presents **S**tring **A**nalysis for **S**ubC (SAS), a tool that statically uncovers *all* string manipulation errors in a subset of C programs. This algorithm cannot handle structures and multi-level pointers.

In many programs, our algorithm is capable of precisely handling destructive memory updates, even in the presence of *overlapping* pointer variables which reference the same buffer at different offsets. This algorithm can uncover defects which go undetected by previous works. Technically, the algorithm is based on a reduction of the problem of checking string manipulation to that of analyzing integer variables.

3.1 Overview

Figure 3.1 shows three interesting erroneous C functions analyzed by our algorithm. Unclean expressions are underlined. Figure 3.1(a) shows an example of overlapping. After statement l_2, `p` overlaps `s` by 5, i.e., `p` points to the sixth character in the string pointed to by `s`. This implies that after statement l_3, which destructively updates `p`, `s` points to a 12-character string and that the expression at l_4 is unclean — it copies 13 bytes (including null) into a 10-byte array. Our static analysis algorithm detects this.

A typical error of accessing an array without bound-checking is shown in Figure 3.1(b). If the array pointed to by `s` does not contain a blank, `s` will be incremented beyond the allocated bounds. Our algorithm analyzes pointer arithmetic and reports a potentially unclean expression at l_2.

It is quite challenging to find unclean expressions in the function `insert_long` shown in Figure 3.1(c) without reporting false alarms. As shown in Section 3.3, our algorithm achieves that. This function inserts the string `"(long)"` at offset `cp` by copying the data up to `cp` into a temporary array, appending the string `"(long)"` and appending the remainder of the string. When invoked, `cp` overlaps `buf`. A buffer overflow occurs at l_3 and/or l_4 if not enough space is left in `temp`. Therefore, these expressions are flagged as potentially unclean. In addition, our algorithm verifies that all the other expressions `buf[i]`, `temp[i]`, and `strcpy(buf, temp)` are clean. Knowing that `buf` and `cp` overlap and analyzing the condition at statement l_1 is necessary in order to validate that the access to `buf[i]` is within bounds, and so is `temp[i]` (since `i` is less than `BUFSIZ`).

A key insight of this algorithm is the *overlapping* property — the number of bytes between two pointer values. The overlapping information enables our method to handle destructive updates in a rather precise way. When updating a pointer value such as *p=0, the overlapping information between `p` and another pointer, say `q`, is used to compute the effect of the update on the length of `q`. Technically, we propose to *reduce* the problem of checking string-manipulation cleanness violations to that of analyzing integer variables — a problem with many known solutions.

```
void simple () {
    char s[20],t[10],*p;

    l1 : strcpy(s,"Hello");
    l2 : p = s+5;
    l3 : strcpy(p, "-world!");
    l4 : strcpy(t,s);
}
```

(a)

```
/* web2c [strpascal.c] */
void null_terminate (char *s) {
    l1 :  while (*s != ' ')
    l2 :          s++;

    l3 :  *s = 0;
}
```

(b)

```
/* web2c [fixwrites.c] */
#define BUFSIZ 1024
char buf[BUFSIZ];

char* insert_long (char *cp) {

    char temp[BUFSIZ];
    int i;

    l1 :  for (i=0; &buf[i] < cp; ++i)
    l2 :       temp[i] = buf[i];

    l3 : strcpy (&temp[i],"(long)");
    l4 : strcpy (&temp[i + 6], cp);
    l5 : strcpy (buf, temp);
    l6 : return cp + 6;
}
```

(c)

Figure 3.1: Unclean string manipulation functions. Unclean expressions are underlined. The function `simple` is from [68] and the other two functions are from `web2c`.

The following subsections elaborate on the main results of this algorithm.

A Source-to-Source C Transformation

We describe a source-to-source program transformation that produces an instrumented C program which dynamically checks that expressions are clean using `assert` statements. For example, the instrumented version of Figure 3.1(a) shown in Figure 3.2 will abort at $cond_4$ indicating that the expression `strcpy(t, s)` at l_4 of the original program is unclean. The details of the transformation technique are explained in Section 3.2.

If the original program is clean, then the instrumented program yields the same results. However, if the original program contains an unclean string expression, then the instrumented program will abort on an `assert` statement, while the result of the original program is undefined.

The idea of instrumenting programs to dynamically check cleanness is not new. For example, SafeC [3] and [45] also instrument the C source with assertions. Purify [35] instruments the executable files. These tools instrument every physical buffer by extending it with additional information and checking this additional information at runtime. Our approach is to instrument the program, adding computations on new integer variables, but analyzing them at compile time.

3.1.1 Applying Integer Analysis

The static problem is to analyze the instrumented program and to mark potentially unclean expressions by detecting that an `assert` statement may not hold. We aim at a *conservative* solution, i.e., one that never misses an unclean string manipulation. Since our goal is to prove that conservative static cleanness checking of realistic C programs is feasible, while generating only a small number of false alarms, we use a rather complex but precise integer analysis that detects linear restraints among the integer variables that our transformation introduces [12]. The integer analysis is conservative thus resulting in an algorithm that detects all unclean string manipulations.

3.1.2 A Prototype Implementation

We analyzed functions with massive string manipulations from three different applications: (i) `fixoutput` — a checker for the output of a lexical analyzer; (ii) `agrep` — a `grep` application; and (iii) `web2c` — a converter from TeX, Metafont, and other related WEB programs to C. A total of 19 errors were detected with only four false alarms.

The rest of this chapter is organized as follows: Section 3.2 describes the source-to-source transformation. Section 3.3 shows how the integer analysis algorithm of Cousot and Halbwachs [12] is used to analyze the instrumented program to find all potentially unclean string expressions in a rather precise way. Finally, Section 3.4 reports more information on the analysis results.

3.2 The Transformation

The instrumented program contains the original program with additional variables and statements. Figure 3.2 shows the instrumented function of Figure 3.1(a). The instrumented code is indented and appears in boldface. For reasons of clarity, we omitted some of the instrumented code and discuss it later in Section 3.2.2.

A *string variable* is either a pointer to `char`, such as `p` in the example, or an array of characters such as `s` and `t`. For every string variable, v, two integer variables are added, *v*`_len` and *v*`_alloc`. The first holds the length of the string v (excluding the null-terminator) and the second holds the allocated size of the buffer pointed-to by v. These integers are initialized depending on the declaration and the initialization of the variable v.

We say that a pointer variable *p overlaps* a pointer variable *q by i* if p points to the same physical buffer as q but at an offset of i bytes, that is, $p = q + i$. Note that if p overlaps q by i then q overlaps p by $-i$. For every pair of string variables, v_1 and v_2, an integer variable v_1**`_overlaps_`**v_2 is added to hold the overlapping index between them. To reduce the number of generated overlap auxiliary variables a pre-transformation analysis is applied to detect variables that cannot overlap, see Section 3.2.1.

In addition to the integer variables, Boolean variables are added to indicate the status of the variable:

- `is_`*v*`_alloc` indicates whether variable v points to an allocated memory region;
- `is_`*v*`_string` indicates whether the data pointed to by v is a null-terminated string; and
- `is_`v_1`_overlaps_`v_2 indicates whether variables v_1 and v_2 overlap.

For every statement, st, there is a cleanness condition that is checked via an `assert` statement prior to statement st. The cleanness condition is generated in a syntax-directed fashion and is labeled as $cond_{st}$.

Every statement st is followed by statements that update the instrumented variables. These are labeled as $post_{st}$. In the example this is rather straightforward. After statement l_1, the length of `s` is updated. After statement l_2, `p_overlaps_s`, the length and the allocated size of `p` are set. After statement l_3 the length of `p` is updated. In addition, since `p` overlaps `s`, the length of `s` is also updated.

Handling function calls is a tricky task. A function call can change the properties of a non-visible variable. For example, a call to a function with an actual parameter x may require updating properties of x (such as x_len). Moreover, due to overlapping, the function may change properties of a variable which is not an argument but which overlaps one. We handle this problem by requiring the contracts to include

```
void simple () {
   char s[20]; int s_len=0, s_alloc=20, is_s_string = false;
   char *p;    int p_len=0, p_alloc=0, is_p_string = false;
   char t[10]; int t_len=0, t_alloc=10, is_t_string = false;
               int p_overlaps_s = 0;

               cond_1 : assert(s_alloc > 5);
   l_1 :  strcpy(s,"Hello");
               post1_1 : s_len = 5;
               post1_2 : s_string = true;
               post1_3 : assert(s_string == true && s_len == 5);

               cond_2 : assert(s_len >= 5);
   l_2 : p = s+5;
               post2_1 : p_overlaps_s = 5;
               post2_2 : p_len = s_len - 5;
               post2_3 : p_alloc = s_alloc - 5;

               cond_3 : assert(p_alloc > 7);
   l_3 : strcpy(p, "-world!");
               post3_1 : p_len = 7;
               post3_2 : p_string = true;
               post3_3 : s_len = p_overlaps_s + p_len;
               post3_4 : assert(p_string == true && p_len == 7);

               cond_4 : assert(t_alloc > s_len);
   l_4 : strcpy(t,s);
               ⋮
   }
```

Figure 3.2: The instrumented function `simple` from Figure 3.1(a). Added instrumentation code is indented and appears in boldface.

```
/* string.h */
char* strcpy(char *dst, const char *src)
   requires   src.string && dst.alloc > src.strlen &&
              is_within_bounds(dst) && !is_overlap(dst,src)
   modifies   *dst
   ensures    dst.strlen == src.strlen &&
              dst.alloc == ⌈dst.alloc⌉_entry &&
              dst.string && return_value== dst ;
   {
     dst_len = src_len;
     dst_string = true;
     FOREACH variable a
          if(is_dst_overlaps_a && dst_overlaps_a >= 0 ) {
            is_a_string = true;
            a_len = dst_overlaps_a + dst_len;
   }
```

Figure 3.3: An executable contract for `strcpy()`.

descriptions of the updates to properties. Technically, this is carried out by requiring an executable code which updates the auxiliary variables. For instance, the contract of the function `strcpy` is shown in Figure 3.3. We refer to these contracts as *executable contracts* as opposed to our fully declarative contract language defined in Section 2.2.

3.2.1 Preliminary Analysis

The number of overlap auxiliary variables added by the transformation can be quadratic in the number of string variables. However, in a typical program the number of variables that do overlap is small. Only assignments and function calls may generate overlapping variables. A preliminary intraprocedural flow-insensitive analysis is performed to detect which variables may overlap. This is done by using a union-find algorithm which assumes that if a may overlap b and b may overlap c then a may overlap c. This algorithm is overly conservative. For example, it assumes that the result value of a function call may overlap any of the input parameters and that every pair of global string variables may overlap. Even so, the experimental results described in Section 3.4 show that the number of overlap auxiliary variables is dramatically reduced.

3.2.2 Transformation of Statements

This algorithm handles a subset of C, called SubC. For every statement, st there are statements that are executed prior to st in order to verify the precondition of st, and statements executed after st to update auxiliary variables.

Next, we describe SubC and formalize the transformation of statements. Function calls are discussed separately.

Statement	Pre Update
$l: v = (*t)$malloc(E) dynamic allocation	assert($E > 0$);
$v = E$ simple assignments	assert(PRE(E));
$*v = E$ destructive update	assert(PRE($*v$)); assert(PRE(E));
free(E) deallocation	assert(IS_ALLOC(E));

Table 3.1: The transformation code of statements' precondition.

Sub C

SubC does not allow multi-level pointers such as in structures (field operator) or due to address-of expression. This is since our algorithm only tracks overlapping of simple variables and not general expressions. Since we track only the number of bytes allocated from a pointer value and not the allocation size prior to the location, subtractions in pointer arithmetic is disallowed. Casting between integer and pointer types is not allowed, since we instrument pointer values and cannot compute this information given an integer. We do allow casting from one pointer type to another pointer type.

Statements' Precondition

The precondition of a statement is verified by an assert statement over the program's original variables and the auxiliary variables. Table 3.1 lists the generated code for the handled set of CoreC statements. This code is inserted before the statement. We only check the precondition of string related violations. Specifically, we check the precondition of pointer arithmetic, references and dereferences as defined in the concrete semantics in Table 2.7

Technically, the precondition is syntactically defined. We use the PRE transformation rule, defined in Table 3.2, which provides the precondition of an expression. Note that not all preconditions as defined in the concrete semantics (see Table 2.7) are verified. On allocations (dynamic and static) we cannot check that the required storage is available. On simple assignments, since the right-hand side has no precondition, we check only the validity of the left-hand side. On destructive updates, both sides have a precondition. For a deallocation, we check that the storage is allocated. This is done by using the `IS_ALLOC` rule defined in Table 3.4.

Table 3.2 lists the precondition of each expression. Simple variables and cast operation have no precondition. On dereference, we verify that the update is within the bounds and within the null-termination byte if it exists. Following our semantics, based on [40, pp.205], we verify that the result of a pointer arithmetic is within bounds or at the first location beyond the rightmost character.

Statements' effect

An execution of a statement can change a property of a variable which is maintained in an auxiliary variable. To this end, the transformation adds code following the statement to apply the necessary changes to the auxiliary variables as defined in Table 3.3.

On allocations, dynamic and static, the allocated size is computed and the relevant `alloc` and `is_alloc` variables are updated according to the allocation.

Exp e	PRE(e)
v variable	true
(t) N cast	true
$*v$ dereference	v_alloc >0 && (!is_v_string || v_len $>= 0$)
$N_1 + N_2$ where: $type(N_1) = t*$ $type(N_2) =$ `int`	!IS_STRING(N_1) && $N_2 * sizeof(t) <=$ ALLOC(N_1) || IS_STRING(N_1) && $N_2 * sizeof(t) <=$ LEN(N_1)

Table 3.2: The PRE transformation rule: preconditions of expressions.

On simple assignment, the properties of the assigned variables are updated according to the properties of the right-hand side. These properties are computed via rules defined in Table 3.4. Assignments create overlapping. We set the overlap auxiliary variable between the left-hand side and the right hand side of the statement and also between any other variables according to the current overlapping.

Destructive updates can come in two different cases: the assignment of ascii zero (a null-terminator) and the assignment of any other character. For assignments of a null-terminator, we update the length of the updated variable. An assignment of a different character may overwrite the existing null-termination byte. A destructive update through one pointer variable can change the properties of overlapping variables.

Function Calls

When handling function calls we need to tackle cases in which the code is not available, e.g., library functions, and also cases of recursive functions. Therefore, function calls are treated by passing additional parameters in a dynamic array pointed-to by a global variable. The array contains the length and allocated size of every string parameter and the overlapping information of every pair of string parameters. Similarly, upon return, the information is passed back along with additional information on the return value and its overlapping with the string parameters. However, a function call may update properties of an invisible variable. For example, say a variable p overlaps variable q is passed as an actual parameter and p's is assigned a string, we need to mark q as pointing to a string. Therefore, we require the contract to include executable code describing the effect of the function call on the auxiliary variables.

Technically, the transformation is as follows: The first statement is an assert statement which verifies the callee's precondition. Next, the additional parameters are assigned into a designated global variable followed by the call itself. Upon return, the additional parameters are returned and the code part of the contract is applied. Last, the postcondition is verified.

Formally, a call statement $N = f(E_1, E_2, ..., E_n)$ where f is declared as

$$
\begin{array}{lll}
t\ f(\cdots) & \textbf{requires} & E_{Pre} \\
 & \textbf{modifies} & E_{M_1}, E_{M_2}, \ldots, E_{M_n} \\
 & \textbf{ensures} & E_{Post}; \\
 & \{S\} &
\end{array}
$$

Statement	Post Update

l: declaration of v
of pointer type $*t$
	is_v_alloc = false;
	v_alloc = 0;
	is_v_string = false;

l: declaration of v
of array type $t[c]$
	is_v_alloc = true;
	v_alloc = $sizeof(t) * c$;
	is_v_string = false;

$l: v = (*t)$malloc(E)
dynamic allocation
	assert(v);
	is_v_alloc = true;
	v_alloc = E;
	is_v_string = false;

$v = E$
simple assignments
	is_v_alloc = IS_ALLOC(E);
	v_alloc = ALLOC(E);
	is_v_string = IS_STRING(E);
	v_len = LEN(E);
	IS_OVERLAP(v, E) = true;
	OVERLAP(v, E) = COMPUTE_OVERLAP(E);
	`FOREACH` variable a
	    is_v_overlaps_a = IS_OVERLAP(v, E) && IS_OVERLAP(E, a);
	    v_overlaps_a = OVERLAP(E, a);

$*v = E$
simple assignments
	if(E == 0) {
	  is_v_string = true;
	  v_len = 0;
	  `FOREACH` variable a
	      if (is_v_overlaps_a && v_overlaps_a $>= 0$) {
	        is_a_string = true;
	        a_len = v_overlaps_a;
	      }
	}
	else if (is_v_string && v_len == 0) {
	  is_v_string = false;
	  `FOREACH` variable a
	      if(is_v_overlaps_a)
	        is_a_string =false;
	}

free(E) deallocation	IS_ALLOC(E) = false;

Table 3.3: The transformation code of statements' effect.

Exp E	**IS_ALLOC(**E**)**	**ALLOC(**E**)**	**IS_STRING(**E**)**	**LEN(**E**)**	**COMPUTE_ OVERLAP(**e**)**
v variable	is_v_alloc	v_alloc	is_v_string	v_len	0
(t) N cast	IS_ALLOC(N)	ALLOC(N)	IS_STRING(N)	LEN(N)	0
$N_1 + N_2$ where: $type(N_1) = t*$ $type(N_2) = \texttt{int}$	IS_ALLOC(N_1)	ALLOC(N_1) - N_2	IS_STRING(N_1) && $N_2 <$ LEN(N_1)	LEN(N_1) - N_2	$N_2 * sizeof\,(t)$

Exp E	**IS_OVERLAP(**v, E**)**	**OVERLAP(**v, E**)**
v' variable	is_v'_overlaps_v	v'_overlaps_v
(t) N cast	IS_OVERLAP(v, N)	OVERLAP(v, N)
$N_1 + N_2$ where: $type(N_1) = t*$ $type(N_2) = \texttt{int}$	IS_OVERLAP(v, N_1)	OVERLAP(v, N_1) + $N_2 * sizeof\,(t)$

Table 3.4: Rules used in the transformation.

Attribute	PRE
E.alloc	ALLOC(E)
E.string	STRING(E)
E.strlen	LEN(E)
overlap(E_1, E_2)	OVERLAP(E_1, E_2)
is_overlap(E_1, E_2)	IS_OVERLAP(E_1, E_2)

Table 3.5: Preconditions of the contract language attributes.

is replaced with the following:

> Assign to temporaries the values of $\lceil E \rceil_{entry}$ expressions appearing in E_{Post}
> `assert`(PRE($E_{Pre}\{E_i/f_i \mid 1 \leq i \leq m\}$));
> assign additional parameters
> $N = f(E_1, E_2, ..., E_n)$
> assign return values to auxiliary variables
> inlined and transformed code of S
> `assert`($E_{Post}\{E_i/f_i \mid 1 \leq i \leq m\}$);

Although the pre- and post-conditions are C expressions, they may use additional attributes. Therefore, we also define transformation rules for contracts' attributes defined in Table 3.5.

Derived Expressions

To enhance the precision of the static analysis the following expressions are replaced by equivalent expressions that use the auxiliary variables:

$*v$	==	0		$\equiv$	(is_v_string && v_len == 0)
$*v$	==	c	(c $\neq$ 0)	$\equiv$	((!is_v_string || v_len > 0) && $*v$ == c)
E_1	-	E_2		$\equiv$	OVERLAP(E_1, E_2)
E_1	>	E_2		$\equiv$	OVERLAP(E_1, E_2) > 0

The first two are conditions relates to the length of the string. Pointer comparisons and subtractions are replaced by expressions over the appropriate `overlap` variable.

3.3 Static Analysis

Our static algorithm detects all the potentially unclean string manipulations. It analyzes the instrumented program and identifies `assert` statements that may be violated. The algorithm is *conservative*, i.e., if there exists an input on which an assertion in the instrumented program is violated, the algorithm will identify this assertion. However, it may yield false alarms by identifying potentially-violated expressions even for assertions in the instrumented program that are never violated.

Our algorithm is iterative. It simultaneously processes Boolean and integer variables by analyzing the program control flow graph and computing conservative information at every node. It then[1] checks if any of the assertions are potentially violated and produces appropriate messages.

To analyze Boolean variables a three-valued domain is used [10]. The value $\top$ results from joining `true` and `false`[2]. The analysis handles Boolean assignments and conditions.

[1]The implementation carries out the check simultaneously.

[2]We chose this solution as it scales better than presenting Booleans as integers.

$$
\begin{aligned}
buf\_alloc &= BUFSIZ \\
temp\_alloc &= BUFSIZ \\
cp\_overlaps\_buf &= BUFSIZ - cp\_alloc \\
cp\_alloc &\geq cp\_len + 1 \\
cp\_alloc &\leq BUFSIZ - 2 \\
cp\_len &\geq 0 \\
i &= 1
\end{aligned}
$$

(a)

$$
\begin{aligned}
buf\_alloc &= BUFSIZ \\
temp\_alloc &= BUFSIZ \\
cp\_overlaps\_buf &= BUFSIZ - cp\_alloc \\
cp\_alloc &\geq cp\_len + 1 \\
cp\_alloc &\leq BUFSIZ - 1 - i \\
cp\_len &\geq 0 \\
i &\geq 1 \\
i &\leq 2
\end{aligned}
$$

(b)

$$
\begin{aligned}
buf\_alloc &= BUFSIZ \\
temp\_alloc &= BUFSIZ \\
cp\_overlaps\_buf &= BUFSIZ - cp\_alloc \\
cp\_alloc &\geq cp\_len + 1 \\
cp\_len &\geq 0 \\
i &\leq BUFSIZ - 1 - cp\_alloc \\
i &\geq 1
\end{aligned}
$$

(c)

Figure 3.4: Application of widening when analyzing the loop in `insert_long`. (c) displays the result of the widening operation applied to (a) and (b), the static information arising at the first and second iterations, respectively.

```
char* insert_long(char *cp)
   requires   cp.string && cp.alloc <= BUFSIZ &&
              is_overlap(cp,buf) && 0 < overlap(cp,buf) < BUFSIZ
   modifies   *cp
   ensures    cp.string && cp.strlen == ⌈cp.strlen⌉_entry + 6 &&
              cp.alloc == ⌈cp.alloc⌉_entry &&
              return_value == ⌈cp⌉_entry + 6 ;
   {
     cp_len = ⌈cp.strlen⌉_entry + 6;
     FOREACH variable a
          if(is_cp_overlaps_a && cp_overlaps_a >= 0 )
            a_len = cp_overlaps_a + cp_len;
   }
```

Figure 3.5: The contract of `insert_long`.

3.3.1 Integer Analysis

We use a linear relation analysis algorithm which discovers linear inequalities among numerical variables [12, 32]. This method identifies linear inequalities of the form: $\Sigma_{i=1}^{n} c_i x_i + b \geq 0$, where x_i is a variable and c_i and b are constants.

The analysis uses the provided set of contracts. The contract of the function `insert_long` (shown in Figure 3.1(c)) is shown in Figure 3.5. This contract is one of the most difficult contracts written in our study. Usually, contracts indicate string data types for parameters and return values. In this example, the precondition indicates that `cp` is a string and its allocation is not greater than `BUFSIZ`. The last two expressions in the precondition concern overlapping and indicate that `cp` may only point to locations within `buf`. The postcondition of this function is obvious.

In the analysis of the function `insert_long`, the algorithm detects the inequalities shown in Figure 3.4(c) upon entry into the loop body (after at least one execution of the loop body[3]). For simplicity, we only show the relevant (in)equalities. The first five (in)equalities are derived from the declarations of the string variables and from the function's contract. The last two inequalities constrain the loop induction variable `i`. Note that `i` is always less than `BUFSIZ` which is important in order to avoid reporting false alarms while accessing `temp[i]` and `buf[i]`.

Figure 3.6 demonstrates how our static algorithm identifies the unclean expression `strcpy(&temp[i], "(long)")` in function `insert_long`. For reasons of clarity, we omitted constraints that are irrelevant to the `assert` verified. From this information, one can understand that when the allocation size of the parameter `cp` is less than or equal to 6, a buffer overflow occurs at l_3. For each potential cleanness violation, the algorithm locates the constraints on the variables (of the instrumented program) where the violation occurs. Of course, in a practical tool these errors need to be reported in terms of the original program.

Technically, linear inequalities are represented by a closed convex polyhedron (a polyhedron, for short) which has two representations: (i) a *system of linear inequalities* $P = \{X | AX \geq B\}$, where

[3] Our simplification step transforms all loops into `do-while` loops with surrounding conditional statements.

l_{3_1} : tmp1 = temp + i ;
$cond_3$: assert(tmp1_alloc $>$ 6)
```
error: the assertion may be violated when:
       i = BUFSIZ - cp_alloc
       tmp1_alloc = cp_alloc
```
cp_alloc $\leq$ 6
l_3 : strcpy (tmp1 ,"(long)") ;

Figure 3.6: A report of a potentially unclean expression `strcpy(&temp[i], "(long)")` in `insert_long` (see Figure 3.1(c)) by the static analysis algorithm. (Expression l_{3_1} is added by the simplifier.)

A is an $m \times n$-matrix, *B* is an m-vector, and n is the number of numerical variables; (ii) a *system of generators* which can be viewed as a geometric representation. See [12, 32] for more information.

Upon termination of the iterative algorithm, the polyhedron at every control flow node conservatively represents the inequalities that are guaranteed to hold whenever the control reaches the respective point.

When two control flow edges are merged, a new polyhedron "including" both polyhedra is computed. Figure 3.7 shows the result of the join operation at l_3 in the analysis of `insert_long`. Figure 3.7(a) shows the polyhedron corresponding to the case where the loop is not executed. In this case, `i` is 0 and `cp_overlaps_buf` is 0, indicating that `cp` and `buf` both point to the same location. Figure 3.7(b) shows the polyhedron corresponding to the end of the execution of the loop. In this case `i` is greater than zero, `cp` and `&buf[i]` point to the same location, which implies that `i` is equal to `cp_overlaps_buf`. The result of the join operation is shown in Figure 3.7(c). The resulting polyhedron includes the case where `i` is at least zero and is equal to `cp_overlaps_buf`.

Loops present a more complex situation. In order to guarantee that the algorithm terminates, a widening operation is used on loop back edges. The resulting polyhedron is defined by those constraints which are satisfied by both polyhedra. Figure 3.4 demonstrates a widening operation at the loop in `insert_long`. Figure 3.4(a) shows the polyhedron describing the first execution of the loop, where `i` equals 1. Figure 3.4(b) shows the polyhedron describing the second execution of the loop, where `i` is between 1 and 2. The result of the widening is shown in Figure 3.4(c) in which the boundaries of `i` are relaxed, `i` is at least 1 and less than `BUFSIZ` $-$ `cp_alloc`.

3.3.2 Soundness

Proving that the SAS algorithm detects all string related violations in the SubC language involves establishment of a number of claims. First, we claim that our transformation is a simulation of the original program. Specifically, the transformation preserves the semantics of clean C programs and the values of certain properties. Next, we claim that our transformation checks the absence of string violations via an `assert` statement. Finally, we claim that the static analysis detects all potential violation of asserts, which is derived from the soundness of the integer analysis (see [12]).

Definition 3.3.1 A *simulation realtion* between a state $\sigma^{\natural}_{tr}$ of the transformation and a state $\sigma^{\natural}$ of the original program, denoted as $(\sigma^{\natural}_{tr}, \sigma^{\natural}) \in \varphi$ is defined as:

$$\begin{aligned} buf\_alloc &= BUFSIZ \\ temp\_alloc &= BUFSIZ \\ cp\_overlaps\_buf &= 0 \\ cp\_alloc &= BUFSIZ \\ cp\_len &\leq BUFSIZ - 1 \\ cp\_len &\geq 0 \\ i &= 0 \end{aligned}$$

(a)

$$\begin{aligned} buf\_alloc &= BUFSIZ \\ temp\_alloc &= BUFSIZ \\ cp\_alloc &= BUFSIZ - i \\ cp\_overlaps\_buf &= i \\ cp\_alloc &\geq cp\_len + 1 \\ cp\_alloc &\leq BUFSIZ - 1 \\ cp\_len &\geq 0 \end{aligned}$$

(b)

$$\begin{aligned} buf\_alloc &= BUFSIZ \\ temp\_alloc &= BUFSIZ \\ cp\_overlaps\_buf &= i \\ cp\_alloc &= BUFSIZ - i \\ cp\_len &\geq 0 \\ i &\geq 0 \end{aligned}$$

(c)

Figure 3.7: The result of join operation at l_3 in the analysis of `insert_long`. (c) shows the result of joining (a) which corresponds to the case in which the loop was not executed and (b) which assumes that the loop was executed.

1. For every expression E of P:
$$[\![E]\!](\sigma^\natural_{tr}) = [\![E]\!](\sigma^\natural)$$

2. For every variable v of P:
$$\begin{aligned} &[\![v\text{\_string}]\!](\sigma^\natural_{tr}) = \mathit{string}^\natural(\sigma^\natural)(v) \\ &[\![v\text{\_len}]\!](\sigma^\natural_{tr}) = \mathit{strlen}^\natural(\sigma^\natural)(v) \\ &[\![v\text{\_alloc}]\!](\sigma^\natural_{tr}) = \mathit{alloc}^\natural(\sigma^\natural)(rv(v)) \\ &[\![v\text{\_is\_alloc}]\!](\sigma^\natural_{\mathrm{tr}}) = \mathit{alloc}^\natural(\sigma^\natural)(rv(v)) > \mathit{0} \end{aligned}$$

3. For every pair of variables v_1, v_2:
$$\begin{aligned} &[\![v_1\text{\_is\_overlaps\_}v_2]\!](\sigma^\natural_{tr}) = \mathit{is\_overlap}^\natural(v_1, v_2)(\sigma^\natural) \\ &[\![v_1\text{\_overlaps\_}v_2]\!](\sigma^\natural_{tr}) = \mathit{overlap}^\natural(v_1, v_2)(\sigma^\natural) \end{aligned}$$

The transformation defined here uses the provided set of executable contracts to update auxiliary variables according to the function. Clearly, when given an incorrect contract our transformation is not a simulation. Therefore, we assume that the contracts are *simulation preserving*.

Definition 3.3.2 A contract of function f is *simulation preserving* with respect to simulation relation φ if for every program P the following holds:

For every pair of states $(\sigma^\natural, \sigma^\natural_{tr})$ arising before call statement S_c to function f in P and $Tr(P)$, respectively such that $(\sigma^\natural, \sigma^\natural_{tr}) \in \varphi$ then:

$$([\![S_c]\!]\sigma^\natural, [\![Tr(S_c)]\!]\sigma^\natural_{tr}) \in \varphi$$

Thus, the state arising after the execution of the transformation of the call, which includes the code of the executable contract, is a simulation of executing the call in the original program.

Lemma 3.3.3 *Given a set of simulation preserving contracts, the transformation defined in Section 3.2.2 is a simulation with respect to simulation relation φ.*

The proof of this lemma is by rule induction on the statements. It is omitted since it is rather a simplified case of a proof of the simulation lemma of the CSSV algorithm.

Lemma 3.3.4 *If there is a string violation in P then $Tr(P)$ arises an assert violation.*

This lemma relies on the fact that all string preconditions are transformed into assert statements. Proving that the SAS algorithm detects all potential string violations in P relies on this lemma and on the soundness of the applied integer analysis.

3.4 Experimental Results

To investigate the usability of our approach in terms of false alarms and cost, we implemented our algorithm for a large subset of C.

3.4.1 Prototype Implementation

Our algorithm was implemented for a subset of C. The first phase is the CoreC simplifier was implemented under SUIF [67]. It generates a new C file.

The pre-transformation step (described in Section 3.2.1) marks which string variables may overlap and generates a control flow graph. Next, an analysis of the *string-related* integer variables is applied. An integer variable i is *string-related* if it is used: (i) as an index to a string variable, e.g., `str[`*i*`]`, (ii) in a pointer arithmetic expression, e.g., `str`+*i*, (iii) in a function call returning a string variable, e.g., `malloc(`*i*`)`, or (iv) as a parameter to a function that also has a string parameter, e.g., `strncpy(dst,src,`*i*`)`.

The static analysis phase is implemented in C using the polyhedra library [37]. The final output of the static analysis is a list of potentially unclean expressions. The analysis provides a description of potentially-violated `assert` statements in the format shown in Figure 3.6. The instrumented program provides a conceptual tool in order to validate whether an error message is a false alarm — assertions which are flagged by the static analysis algorithm but cannot be violated by any program input.

When our static analysis algorithm identifies an assertion violation, it flags an error (in the form shown in Figure 3.6). After reporting the error it (optimistically) assumes that the assertion condition holds on subsequent execution paths to avoid repetitive error messages. This is in line with the instrumented program that aborts at the first violated assertion. For example, our algorithm does not flag the expression `cp+6` at l_6 in Figure 3.1(c) as unclean. This expression is unclean when the allocation size of the buffer pointed to by `cp` is less than 6. In this case, `strcpy(&temp[i],"(long)")` at l_3 is also unclean and must be executed prior to l_6. However, both expressions in l_3 and l_4 are detected as unclean since they can be violated on different inputs.

3.4.2 Results

Table 3.6 presents the analysis results. Columns 3–5 provide source information: (i) number of lines of simplified code, (ii) number of nodes in the analyzed CFG, and (iii) number of string variables in

App.	Function	Source code info				Static Analysis				Messages	
		LOC	CFG	Strings	Annotations	Overlap	Vars	CPU sec	Space MB	False Alarms	Errors
fixoutput	flush	19	11	6	BO	15	30	0.4	1.2	0	0
	getchar	33	31	8	BOI	15	36	3.0	5.1	0	3
	backup	22	28	6	BO	6	22	0.6	0.7	0	0
	getstr	50	42	7	BOI	6	28	8.0	9.2	0	0
	main	403	157	10	-	10	44	309.9	99.5	0	0
agrep	extend_re	30	14	11	SB	28	54	3.2	9.6	0	0
	init	45	19	3	S	1	9	0.3	0.3	0	0
	coutline	22	12	10	SBI	28	61	1.4	4.4	0	0
	m_short	187	109	14	BI	78	125	279.8	242.7	0	5
web2c	fprint_pascal_string	29	10	3	S	1	9	0.1	0.1	0	2
	null_terminate	11	7	2	S	0	5	0.1	0.1	0	2
	space_terminate	12	9	2	S	0	5	0.1	0.1	0	0
	extendfilename	17	13	4	S	1	11	0.3	0.6	0	2
	remove_newline	18	11	12	S	29	56	0.8	3.4	0	0
	insert_long	21	25	14	SBOI	78	110	19.6	38.8	0	2
	join	44	27	13	SB	29	60	8.9	16.0	3	1
	skip_balanced	72	24	9	SBO	28	53	2.8	5. 2	1	0
	bare	85	57	9	SB	28	53	13.8	17.1	0	2

Table 3.6: The experimental results. The column named "Annotations" indicates the level of difficulty of annotating the function.

the function including global variables. The ratio between LOC and CFG indicates the fragment of the program analyzed. In some cases, such as `backup` and `insert_long`, all statements are string manipulations. In others, such as `main`, a small fraction is analyzed.

Column 6 indicates the level of difficulty of annotating the functions. We use the characters 'S','B', 'O' and 'I' as follows: (S) for simple specification, such as `string` and `is_within_bounds`, (B) for specifying the boundaries of buffers, (O) for specifying overlapping information, and (I) for other integer relations. There was no need to provide pointer specification in the analyzed code.

Columns 7–10 show some statistics on the analysis: (i) the number of overlapping variables[4]; (ii) the total number of integer variables in the analysis: it includes two variables per string, the string-related integer variables, and the overlap variables; (iii) the elapsed CPU time of the analysis in seconds on a Pentium 366 with 128MB running Windows 2000; and (iv) the maximum size of allocated memory during the analysis in MBytes.

From a user point of view, the most important information appears in columns 11–12 that show the number of false alarms vs. the number of errors found. We verified errors by running the program with

[4]The maximum number of overlapping variables is $n * (n-1)/2$, where n is the number of string variables. Only `flush` requires that many variables.

an appropriate input. A total of 19 errors were detected. Ten errors are due to unsafe pointer arithmetic resulting in a pointer beyond the bounds of the buffer. These errors are due to assumptions that a certain character appears in the string, such as in the function `null_terminate` shown in Figure 3.1(b) which assumes that there is a blank in the string. Five errors are due to unsafe calls to standard C functions, such as the calls to `strcpy()` in `insert_long`. The remaining four errors are updates of a single character out of the buffer bounds, such as the expression `*s = 0` in `null_terminate`.

In the function `join` the analysis produced three false alarms due to a complex condition in a `for` loop. In this case the widening operation causes loss of important constraints. The false alarm in `skip_balanced` occurs because it safely assumes that it is invoked with a balanced number of parentheses in the input parameter. This is verified in a prior call to a different function. This example demonstrates that in some cases it is hard to separate cleanness from correctness — in order to show that this function is clean we need to verify correctness, i.e., that the implementation correctly checks that the input string contains a balanced number of parentheses. Fortunately, in most of the analyzed examples, this is not the case, i.e., cleanness does not depend on correctness.

3.5 Conclusions

This algorithm shows that real software defects can be identified by conservative static analysis tools in a realistic subset of C. The analysis produces very few false alarms. It assures against certain defects and the user is relieved from the need to test against them. The analysis combines pointer and integer analysis resulting in a rather precise information.

The method presented in this chapter applies to a large subset of C. Furthermore, general pointer arithmetic can be handled by additional tracking of the allocation size preceding the location pointed-to by a pointer. However there are important widely used C constructs that are difficult to handle using this method. The most problematic construct is multi-level pointers. This comes from the fact that we track properties of locations directly pointed-to by pointer variables. In contrast, checking the cleanness of multi-level pointers require tracking the properties of a location regardless of the variables pointing-to it.

An additional limitation is that the contracts are not verified by the tool since they are executable. Thus, the soundness depends on the correctness of the contracts.

Chapter 4

CSSV: C String Static Verifier

This chapter presents CSSV, a tool that statically uncovers *all* string manipulation errors. Being a conservative tool, it reports all errors of this type at the expense of sometimes generating *false alarms*. Fortunately, only a small number of false alarms are reported, thereby proving that statically reducing software vulnerability is achievable. CSSV handles large programs by analyzing each procedure separately. To this end procedure *contracts* are allowed which are verified by the tool. However, unlike the SAS algorithm, CSSV verifies each contract against the procedure code. Thus, when given an incorrect contract CSSV reports an error.

4.1 Overview

Being a tool conservatively handling full C, CSSV must conclude information regarding pointer interactions. Needless to say, it is difficult for programmers to define contracts in terms of pointer behavior. Therefore, CSSV does not require contract information regarding pointers. Instead, CSSV applies a whole-program flow-insensitive pointer analysis to detect statically which pointers may point-to the same base address. Conducting pointer analysis in a language like C is a non-trivial task. Fortunately, several flow-insensitive algorithms have been shown to run on whole applications of considerable size, e.g., [14, 33]. However, the precision of these algorithms is somewhat low since they combine information from different states of the program. CSSV then applies an algorithm that extracts *procedural points-to information* for the analyzed procedure *P*. Our algorithm benefits from the fact that memory locations inaccessible from visible variables of *P* cannot affect the postcondition of *P*. In many cases this allows subsequent analyses to perform strong updates [7] when analyzing the procedure's body. We also compute certain *must-aliases* to improve the precision of the whole-program flow-insensitive pointer analysis.

The procedure code and points-to information are fed into the *C2IP* transformer. C2IP generates a procedure that manipulates integers. C2IP guarantees that if there is a runtime string manipulation error in a procedure invocation then either (i) the procedure's precondition did not hold on this invocation, or (ii) an `assert` statement in the resultant integer program is violated on a corresponding input. In addition, C2IP checks pointer assertions if specified in the contracts.

Finally, the resultant integer program is analyzed using a conservative integer-analysis algorithm to determine all potential violations of `assert` statements. Because the integer and pointer analyses are sound and because contracts are verified both at call sites and at the procedural level, all string errors are reported. In particular, the integer analysis reports an error when the specified postcondition is not guaranteed to hold. For minimizing the number of false alarms, CSSV uses a rather precise integer analysis that represents linear relationships on integer variables. The final result is a list of potential errors. For every error, a counter-example is generated that can assist the programmer in determining if a message is a real error or a false alarm. False alarms may occur due to (i) erroneous or overly weak contracts, (ii) abstractions conducted by C2IP, or (iii) imprecision of the pointer or integer analyses.

The rest of this chapter is organized as follows: Section 4.2 describes the pointer analysis step of CSSV. Section 4.3 describes the C2IP transformation. The soundness of CSSV is proven in Section 4.4. Empirical results are presented in Section 4.5. Finally, Section 4.8 concludes this chapter.

4.2 Procedural Pointer Analysis

Defining contracts concerning pointer behavior is a difficult task for programmers. Fortunately, several flow-insensitive algorithms have been shown to run on whole applications of considerable size, e.g., [14, 33], hence omitting the need for contracts regarding points-to information. In particular, for every function, these algorithms provide a summary of all of its calling contexts. These algorithms, based on the theory of abstract interpretation, provide sound and finite representations even in the presence of recursive functions or nonterminating programs. In principle, a conservative analysis can utilize this information and analyze a function with all possible calling contexts. However, this can yield many false alarms. For example, if we analyze function `SkipLine` (shown in Figure 2.2) with its two calling contexts, the pointer analysis will yield that `PtrEndText` may point to either `s` or `r`. Hence, the updates to `*PtrEndText` will be treated as weak updates, i.e., taking into account that the store may or may not change. Therefore, the analysis will fail to show that the postcondition holds. As a result, a false alarm will be issued. CSSV avoids this false alarm by performing strong updates in certain cases. The main idea is to precompute *procedural points-to information* that guarantees that strong updates to the properties of `*PtrEndText` can be performed. In general, it guarantees that in well-behaved programs direct updates through the formal parameters can be interpreted as strong updates.

This section starts by presenting two observations: the *base and offset* observation and the *summary* observation. These two important observations related to the use of pointer analysis for static analysis in general and for error detecting tools specifically.

The Base and Offset Observation

One of the key ideas behind the analysis is to describe a pointer as a pair of *(base, offset)* where the *base* is the base address of the location pointed-to by the pointer, and the *offset* is the number of bytes from the base to the location. The theoretical and technical advantages of this representation are:

- Although the exact values of locations and base addresses are ignored, this representation abstracts the exact location within an "object" a pointer points-to. The representation maintains *abstract locations* representing base addresses and the possible offsets of a pointer.

- Similar to many pointer analysis, the (base, offset) representation can reason about *aliases* — two pointers pointing to the same location. Moreover, we can reason about *overlapping* — two pointers pointing within the same base address but at different offsets. This is useful is many cases. For example, in the contract of the `strcpy()` library function, shown in Figure 2.4, it is required that the arguments do not overlap.

- Existing pointer analysis techniques can compute the *base* part of pointers.

- As we shall see, existing integer analysis techniques can be adapted to detect the *offset* part of pointers.

It is interesting to note that runtime verification detecting algorithms also maintain pointers as pairs of base and offset. Among them are CCured [50] and SafeC [3]. It might be surprising that our semantics defined in the previous chapter does not follow this approach. Instead of representing pointers as pairs, we maintain for each location its base. Indeed, our semantics is simple, clean and handles whole C. One cannot handle casting with only the (base, offset) representation. Although a cast from a pointer to integer can be easily computed by adding the base and offset, a cast from an integer value to a (base, offset) cannot be computed.

Unlike the concrete world, in the abstract world we aim at a representation that in most common cases yields a precise answer and in some cases returns an "I don't know" answer. In the case of pointer-integer casting our abstraction yields such an answer. A cast from integer to pointer yields that the pointer may point to any base address with any offset and it may even be to an unallocated location. Similarly, a cast from pointer to integer yields that the integer can have any value.

The Summary Observation

As we shall see, we compute when an abstract location can represent multiple locations in a single concrete state. This is called *summary abstract location*. Many conservative applications of points-to analyses do not require summary information and perform weak updates when destructive update statements, e.g., `*p = e`, are analyzed. Our analysis reduces the number of false alarms by tracking summary properties, and uses strong update when possible.

4.2.1 Abstract Points-to States

Two abstract states representing potential pointer relationships in concrete states are presented. The first, abstracts all pointer-relations during the execution of a whole program while the second focuses on the pointer values during the execution of a given procedure. In both states, a pointer is represented by a pointer from the base of the source to the base of the target.

Procedural Abstract Points-To State

The *procedural points-to* information is an abstraction of all potential relationships between base addresses of locations that may point to each other in concrete states that occur during the execution of procedure P.

Definition 4.2.1 A **procedural abstract points-to state of** P (PPT) is a tuple

$$\sigma^P = (\mathcal{BA}_P, loc_P, pt_P, count_P, allocatedAt_P)$$

where:

- $\mathcal{BA}_P$ is a set of abstract locations.
- $loc_P : visvar_P \rightarrow 2^{\mathcal{BA}_P}$ maps a variable into a set of abstract locations representing the variable's global or stack locations.
- $pt_P : \mathcal{BA}_P \rightarrow 2^{\mathcal{BA}_P}$ abstracts the pointers from one location to another.
- $count_P : \mathcal{BA}_P \rightarrow \{1, \infty\}$ abstracts the number of concrete base addresses represented by an abstract location. It represents summary information as discussed above.
- $allocatedAt_P : \mathcal{BA}_P \rightarrow$ Lab abstracts the allocation label (statement) of the base addresses represented by an abstract location.

We shall first establish a number of intended properties of a PPT:

$\mathcal{BA}_P$ abstracts reachable base addresses. However, as we shall see, an abstract location represents values and properties of all locations of the base address.

loc_P is usually a singleton representing the single and unique stack, static or global location of a variable. In cases of recursive functions the mapping may be to a set of abstract locations.

pt_P abstracts pointers from base addresses and locations. A pointer from a location $l^\natural$ is represented by a pointer from the abstraction of a base address of $l^\natural$.

$count_P$ maps to ∞ when the abstract location may represent more than one base address in a given concrete state. Thus, the mapping is one when the abstract location is guaranteed to represent at most one base address. An abstract location having $count_P = \infty$ is a called **summary abstract location**. Summary abstract locations are used to finitely represent unbounded sets of base addresses.

$allocatedAt_P$ maps to a single label. Thus, base addresses allocated at different labels are represented by different abstract locations.

Global Abstract Points-To State

As mentioned, a global abstract points-to state which summarizes all calling contexts may not be precise enough for an effective cleanness checking tool due to the many false alarms. However, the global information is needed to compute the procedural information. The definition of a global abstract points-to state is similar to the procedural one. However, the state abstracts all potential pointer relationships between locations during the execution of the program. In the global view all variables of the program are of interest.

Definition 4.2.2 A *global abstract points-to state* (GPT) is a tuple

$$\sigma^G = (\mathcal{BA}, loc, pt, count, allocatedAt)$$

where:

- $\mathcal{BA}$ includes all abstract locations.
- $loc\colon var \rightarrow 2^{\mathcal{BA}}$.
- $pt\colon \mathcal{BA} \rightarrow 2^{\mathcal{BA}}$.
- $count\colon \mathcal{BA} \rightarrow \{1, \infty\}$
- $allocatedAt\colon \mathcal{BA} \rightarrow \text{Lab}$

4.2.2 Soundness of Abstract Points-to State

To generalize the following definition and lemma, we use the name abstract points-to (APT) state,

$$\sigma^{APT} = (\mathcal{BA}, loc, pt, count, allocatedAt)$$

to refer to both global and procedural abstract points-to states.

Definition 4.2.3 An abstract points-to state

$$\sigma^{APT} = (\mathcal{BA}, loc, pt, count, allocatedAt)$$

is a *sound approximation of a concrete state* $\sigma^\natural$ and a set of variables *var* with respect to an abstraction function $\alpha\colon \mathcal{L}^\natural(\sigma^\natural) \rightarrow \mathcal{BA}$ if:

1. Sound and distinct abstraction of *loc*:
   - For all $v \in \mathit{var}, i \in \mathbf{N}$: $\alpha(\mathit{loc}^\natural(\sigma^\natural)(v,i)) \in loc(v)$.
   - For all $v_1, v_2 \in \mathit{var}, i, j \in \mathbf{N}$: $v_1 \neq v_2 \Rightarrow \alpha(\mathit{loc}^\natural(\sigma^\natural)(v_1,i)) \cap \alpha(\mathit{loc}^\natural(\sigma^\natural)(v_2,j)) = \emptyset$
2. Same representation of all locations of a base address:
   For all $l^\natural \in \mathcal{L}^\natural$
   $$\alpha(l^\natural) = \alpha(\mathit{base}^\natural(\sigma^\natural)(l^\natural))$$
3. Representing all pointer values:
   For all $l_1^\natural, l_2^\natural \in \mathcal{L}^\natural(\sigma^\natural)$ such that $l_1^\natural$ and $l_2^\natural$ are reachable from some $v \in \mathit{var}$ satisfying
   $$\mathit{content}(\sigma^\natural)(l_1^\natural, \texttt{sizeof\_ptr}) = l_2^\natural\colon$$
   then
   $$\alpha(l_2^\natural) \in pt(\alpha(l_1^\natural))$$
4. Sound summary abstract locations:
   For all $b_1^\natural, b_2^\natural \in \mathcal{BA}^\natural(\sigma^\natural)$ such that $b_1^\natural \neq b_2^\natural$ and $\alpha(b_1^\natural) = \alpha(b_2^\natural)$, then
   $$count(\alpha(b_1^\natural)) = \infty$$
5. Sound representation of allocation site
   For all $b^\natural \in \mathcal{BA}^\natural$:
   $$\mathit{allocatedAt}^\natural(\sigma^\natural)(b^\natural) = allocatedAt(\alpha(b^\natural))$$

Based on this definition, we define the soundness of a GPT for program fragments. We use two definitions: (i) *sound approximation* of all admissible states and (ii) *sound partition* which requires also that the memory locations be partitioned into abstract locations, thus that a location is mapped to the same abstract location in following steps of execution.

Definition 4.2.4 An abstract global points-to state, σ^G, is a *sound approximation of program fragment* P if it is a sound approximation of all states $\sigma^\natural \in \Sigma^\natural(P)$ which arise in the execution of P.

Expression		PT-L-Value
v	scalar variable	$loc(v)$
$*N$	simple dereference	$pt(lv_{pt}(N))$
$E.f$	all field operator expressions	$lv_{pt}(E)$

Table 4.1: PT-L-value of expressions in an abstract points-to state $\sigma^{APT} = (\mathcal{BA}, loc, pt, count, allocatedAt)$.

Definition 4.2.5 An abstract global points-to state, σ^G, is a *sound partition of program fragment P* if the following requirements holds:

Sound Approximation σ^G it is a sound approximation of P.

Partitioning For each state $\sigma^\natural \in \Sigma^\natural(P)$ and for each α mapping such that σ^G is a sound approximation of $\sigma^\natural$ with respect to α, then for each statement S and state $\sigma'^\natural$ obtained from applying S to $\sigma^\natural$ ($\sigma'^\natural = [\![S]\!]\sigma^\natural$) there exists an α' mapping such that:

- σ^G is a sound approximation of $\sigma'^\natural$ with respect to α'.
- For each $l^\natural \in \mathcal{L}^\natural(\sigma^\natural) : \alpha'(l^\natural) = \alpha(l^\natural)$.

This definition relies on our semantics which never deallocates locations, $\mathcal{L}^\natural(\sigma^\natural) \subseteq \mathcal{L}^\natural(\sigma'^\natural)$. Hereby ensuring that locations are not reused due to deallocation and reallocation as different objects. Locations are allocated at declarations and by calling dynamic memory allocation routines but during free and end-of-scope statements the semantics only indicates that the size is zero.

The existing points-to algorithms of [1, 22, 14] yield sound partitions of a given program P, when summary information is conservatively added. Abstract locations representing dynamically allocated locations are indicated as summary. Abstract locations which are known to represent at most one reachable instance of a stack or automatic locations are non summary. Such information can be obtained using an escape analysis. The pointer analysis techniques described in [15, 59] yield sound approximations but not sound partitions.

4.2.3 Meaning of Expression

In the abstract world, L-values and R-values are defined to return sets of possible values. An L-value (R-value) of an expression E represents all the possible L-values (R-values) of E at all concrete states represented by the abstract points-to state. The abstract points-to state contains information regarding the base part of pointer values. Furthermore, the points-to state cannot determine possible values of numeric expressions. Therefore, this partial information is used to compute the points-to meaning of expressions: *PT-L-values* and *PT-R-values* denoted as $lv_{pt}(e)$ and $rv_{pt}(e)$, respectively.

Table 4.1 defines the PT-L-values of expressions given an abstract points-to state. For scalar variables the PT-L-values is the *loc* mapping. For dereference expressions, the *pt* mapping provides the PT-L-value,

Table 4.2 lists the PT-R-values of pointer-expressions. The result of pointer arithmetic `p + i` yields the abstract locations pointed-to by `p`. This stems from the fact that an abstract location represents all locations within the same base address. In particular, the result of a pointer arithmetic must be within the bounds of the base address pointed-to by `p` and therefore to a location represented by the same abstract

Expression		PT-R-Value
$E_1 \ op_b \ E_2$	pointer arithmetic	$pt(E_1)$
$\&E$	all address-of expression	$lv_{pt}(E)$
$(\ t\ )\ E$	cast	convert $rv(E)$ to type t
E	any other pointer expression	$pt(lv_{pt}(E))$

Table 4.2: PT-R-value of expressions in an abstract points-to state $\sigma^{APT} = (\mathcal{BA}, loc, pt, count, allocatedAt)$.

base address. The other types of expressions have similar definitions as in the concrete semantics. For an address-of expression, the PT-R-value is the PT-L-value of the argument. For other expressions, the PT-R-value is computed using the *pt* mapping and the PT-L-value of the expression. Most attributes in the contract language are integer properties, and therefore do not have a PT-R-value. The only exception is the E.base attribute whose PT-R-value is $pt(rv_{pt}(E))$.

Lemma 4.2.6 *If an abstract points-to state $\sigma^{APT} = (\mathcal{BA}, loc, pt, count, allocatedAt)$, is a sound approximation of program fragment P then:*

for each admissible concrete state $\sigma^{\natural}$ arising during the execution of P and for each expression $E \in$ **Expr***:*

$$\alpha(lv(E, \sigma^{\natural})) \in lv_{pt}(E)$$

The proof of this Lemma appears in Appendix A.1.

4.2.4 Constructing PPT from GPT

The goal in constructing a PPT is to represent the relationships between visible variables of P while ignoring the exact values of the formal parameters of P. For example, the fact that the formal parameter `PtrEndText` of `SkipLine` may point to either `s` or `r` is ignored. Nevertheless, it is important, for soundness reasons, to track potential aliasing between parameters and/or global variables.

Figure 4.1 demonstrates the result of constructing a PPT from a GPT for the procedure `SkipLine`. For clarity, nodes not affected by the process are not represented. When possible we denote abstract locations as either a PT-L-value (e.g., $lv_{pt}(s)$) or as a PT-R-value (e.g., $rv_{pt}(PtrEndText)$) of an expression. We can see that a new abstract location $rv_{pt}(PtrEndText)$ represents the (unique) concrete location which holds the value of `*PtrEndText`.

Given a sound global abstract pointer state of the whole program

$$\sigma^G = (\mathcal{BA}, loc, pt, count, allocatedAt)$$

we will construct a PPT for P:

$$\sigma^P = (\mathcal{BA}_P, loc_P, pt_P, count_P, allocatedAt_P)$$

The mapping loc_P is computed by projecting loc to the visible variables of P. Similarly, $\mathcal{BA}_P$,$count_P$, and $allocatedAt_P$ are computed by including abstract locations reachable from visible variables of P. An initial value for pt_P is obtained by projection. In our running example, this yields the global points-to information without $lv_{pt}(buf)$. We aim at a potentially more precise representation. A conservative algorithm to check if it is sound to merge the nodes $l_1, l_2, \ldots, l_m$ pointed-to by a formal f without

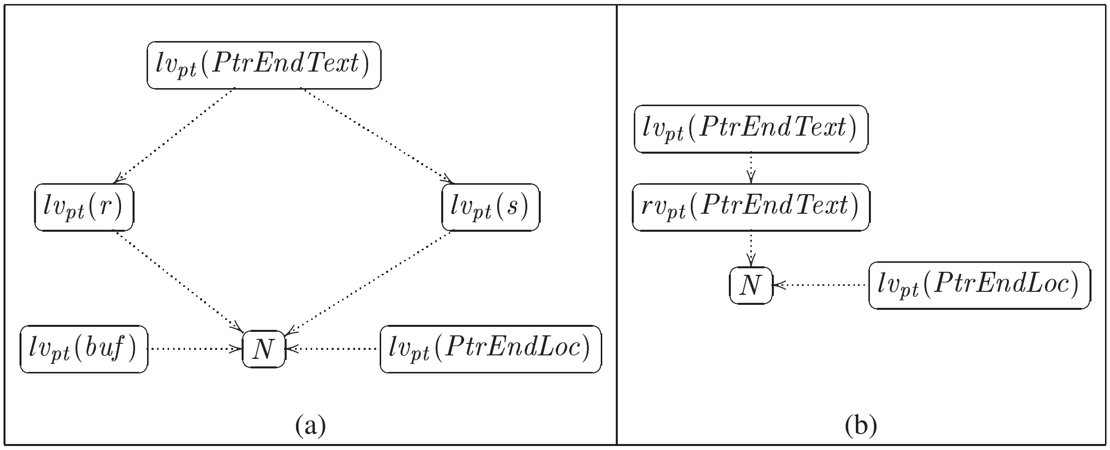

Figure 4.1: The whole-program points-to information for `SkipLine` (a), and the PPT for `SkipLine` (b).

creating a new summary node is presented in Figure 4.2. This algorithm checks that in every concrete state at most one concrete location is represented by $rv(f)$ (the set of abstract locations pointed-to by a formal parameter f). This algorithm may improve the precision of destructive updates through f, but may decrease the precision of other updates. It appears that for modular programming where external aliasing between global and formal variables is limited, this algorithm improves the precision of the static analysis.

If the algorithm concludes that it is safe to represent all nodes by one node, then the merge is done by:

1. replacing $l_1, l_2, \ldots, l_m$ by a single non-summary abstract location $rv_{pt}(f)$.
2. setting $pt(r_f)$ to $\bigcup_{i=1}^{m} pt(l_i)$.

Lemma 4.2.7 *For every formal parameter f and an admissible state at P, $\sigma^\natural$, and a sound approximation of $\sigma^\natural$,*

$$\sigma^P = (\mathcal{BA}_P, loc_P, pt_P, count_P, allocatedAt_P)$$

with respect to a mapping function $\alpha\colon \mathcal{BA}^\natural(\sigma^\natural) \rightarrow \mathcal{BA}_P$, if `Parameterizable`$(\sigma^P, f)$ *returns true then at most one of the reachable locations in $\mathcal{L}^\natural(\sigma^\natural)$ is represented by the abstract locations pointed to by f.*

The proof of this lemma appears in Appendix A.2. It shows that the generated PPT is a sound abstraction.

4.3 C2IP — C To Integer Program

The goal of our abstraction is to detect string related violations. Technically, this abstraction explicitly tracks the possible string lengths, the allocation size of buffers and the existing of the null-termination byte. One of the key elements to the precision of the abstraction is the tracking of the *offset* part of pointers. As discussed in the previous chapter, the *offset* of a pointer p, is the displacement (in bytes) of the location p points to from its base address. The offset property enables the precise handling of pointer arithmetic and destructive updates. Moreover, the offset is maintained at the abstraction representing the L-value of the pointer yielding more precision than abstracting the base address mapping.

```
Boolean parameterizable( PPT σ^P, formal f )
{
  let σ^P = (BA_P, loc_P, pt_P, count_P, allocatedAt_P)
  if not Unique(f) return false
  let l_0 = loc(f)
  let {l_1, l_2, ..., l_m} = pt(l_0)
  for i = 1 to m
      if count(l_i) = ∞ return false
      remove from pt edges from l_0 to l_j where j ≠ i,
      and let pt' be the resultant points-to map.
      if there exists a reachable node l_j, j ≠ i in pt' then
        return false
  // at most one of l_i is reachable in a given concrete state.
  return true;
}
```

Figure 4.2: Algorithm to conservatively check that at most one concrete location is represented by the set pointed-to by a formal parameter f. The function `Unique` checks if $lv_{pt}(f)$ is a single non-summary abstract location.

The above properties (length, allocation size, and offset) are numeric properties which are often related to other scalar integers in the program. We believe that complicated integer relationships are important for the analysis of strings. Furthermore, we believe that in many cases the pointer relationships are rather simple. For example, in the function `SkipLine` there is a numeric relationship between the offset of `PtrEndText`, the allocation size of the location pointed-to by `PtrEndText`, and the integer variable `NbLine`. There are no summary nodes and there is only one overlapping relationship between `*PtrEndText` and `PtrEndLoc`. Therefore, we propose to use the flow-insensitive procedural pointer information, presented in the previous chapter, with a rather precise integer analysis that computes integer inequalities among the numeric properties.

The main idea is a transformation that maps integer properties of interest to *constraint variables* and generates an integer program manipulating the constraint variables. This is done by "summarizing" the properties of all locations represented by the same abstract location in the *PPT*. This process has to handle the following complications:

- The *PPT* contains may information. Thus, a pointer p may point to location $l_1^\natural$ represented by b_1 or to location $l_2^\natural$ represented by b_2. For this, the transformed program represents dereference expressions as sets of integer expressions. For example, the expression $*p$ is transformed to two expressions one over b_1, representing the value in $l_1^\natural$, and the second over b_2, representing the value in $l_2^\natural$. For ease of generating the integer program, we define sugared constructors over sets of expressions.

- The *PPT* represents locations with the same base address by a single abstract location. Thus, a single constraint variable v is used for representing a numeric property of different locations represented by b. The transformation uses the $\top$ value to indicate that any value is possible.

- The *PPT* contains summary abstract locations. A summary abstract location represents one or more base addresses. Each update to a summary abstract location is conservatively treated as *weak update* reflecting the fact that property of one base address may be changed while others' do not change.

We use a standard relational semantics for integer guarded programs. A state consists of a set of environments where each variable is mapped to one value. This tracks relations between variables, i.e., variable v_1 is mapped to c_1 only when v_2 is mapped to c_2. The state can be viewed as points in an n-dimensional space where each variable is mapped to an axis and there are n variables [30].

The integer program is analyzed to detect potential violations of preconditions. The analysis maintains inequalities among the variables, representing relations among the integer properties of base addresses and integer variables. The integer analysis abstract state can be viewed as a closed convex polyhedra containing all points of the represented states of the integer program.

The static analysis defined here does not aim at detecting all aspects of violations of preconditions as defined by the semantics. Concentrating on one type of violations allows us to design an algorithm that tracks the important properties and runtime values related to this type of error while ignoring other values. This algorithm verifies only preconditions related to string violations. Formally, the algorithm detects all violations of dereferences and pointer arithmetic's preconditions and verifies the contracts of procedures. Other violations can be checked by a separate tool which is beyond the scope of this thesis.

The rest of this section is organized as follows: First, Section 4.3.1 introduces the notion of *constraint variables* which abstract numeric properties. Section 4.3.2 defines the guarded integer program and informally defines their semantics. Section 4.3.3 describes the transformation form C to guarded integer programs. Section 4.3.4 briefly sketches the integer analysis phase. Section 4.4 proves the soundness of our algorithm.

4.3.1 Constraint Variables

Constraint variables are used to maintain possible values an integer variable or propriety can have at runtime. The idea is to "attach" to each abstract location a designated set of constraint variables.

Definition 4.3.1 *The set of* **constraint variables** *of a procedure P, denoted as $ConVar_P$ are as follows. For every abstract location $b \in \mathcal{BA}_P$ where $PPT = (\mathcal{BA}_P, loc_P, pt_P, count_P)$ is the procedural points-to of procedure P, the constraint variables are:*

b.***val*** *holds potential primitive values stored in locations represented by b.*

b.***offset*** *holds potential offsets of pointers represented by b.*

b.***size*** *holds allocation sizes of base addresses represented by b.*

b.***string*** *holds when the represented base addresses contain a null terminated string.*

b.***len*** *holds the lengths of the strings (excluding the null byte) of all base addresses represented by b.*

Pre values *In addition a constraint variable "E".Pre is defined for every pre expression $\lceil E \rceil_{entry}$ in P's postcondition or in the postcondition of procedures called by P.*

A constraint variable `return_value`$_X$ *is declared for each procedure X called by P.*

The *PPT* for the `SkipLine` function is shown in Figure 4.3. For the six abstract nodes in the *PPT* we generate 30 constraint variables. In practice, many of them are never used.

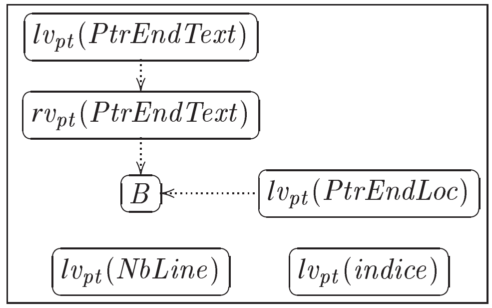

Figure 4.3: The *PPT* for `SkipLine` where each abstract location is given a name.

4.3.2 Guarded Integer Language

The generated integer program is written using a primitive guarded-command language which is a variant of Dijkstra's guarded command [17].

cmd	::=	x:= *exp*	assign to x the value of *exp*
	|	skip	skip
	|	goto l	goto
	|	l: *cmd*	labelled command
	|	if(*exp*) cmd_1 else cmd_2	conditional statement
	|	assert(*exp*)	check that inequality *exp* holds
	|	assume(*exp*)	assume that inequality *exp* holds
	|	$cmd_1; cmd_2$	concatenation, execute cmd_2 after cmd_1
	|	$cmd_1 \,\square\, cmd_2$	nondeterminism — either cmd_1 or cmd_2 is executed
exp	::=	c	constant
	|	$exp_1 \cup exp_2$	set union operation
	|	$exp_1 \setminus exp_2$	set minus operation
	|	$\top$	any value
	|	x	variable
	|	$exp_1\ op_b\ exp_2$	binary expression
	|	op_u exp	unary expression
	|	$b\ ?\ exp_1 : exp_2$	conditional value exp_1 when b holds otherwise exp_2

The assert command terminates the program abnormally. The assume command terminates the program normally when the required condition does not hold. The main difference between `assert` and `assume` is in the analysis. The analysis verifies `assert` statements while for an `assume` statement the argument is regarded as true and thus the analysis refines the abstract state. The $\top$ expression stands for any value.

Sugared Constructs

For ease of presentation, we allow the following sugared constructs:

Numeric operations: All unary, binary and relational operations are extended to sets in a pointwise manner. For example, $E\ op\ E'$ is the set of all expressions $\{e_i\ op\ e'_j \mid e_i \in E, e'_j \in E'\}$.

Set assignment $X = E$ is a shorthand for $\square_{x_i \in X, e_j \in E} x_i = e_j$.

Set if if(E) *cmd*$_1$ else *cmd*$_2$ is a shorthand for $\Box_{e\in E}$ if(e) *cmd*$_1$ else *cmd*$_2$.

Set assert: Assert(E) is a shorthand for assert($\bigwedge_{e_i\in E} e_i$).

Set assume: Assume(E) is a shorthand for assume($\bigvee_{e_i\in E} e_i$).

Modify: modify(x) assigns any value to x. It is a sugar for $x := \top$.

Set modify: Modify(X) where X is a set of variables $x_1, x_2, \cdots, x_n$ is a shorthand for modify(x_1); modify(x_2); $\cdots$ modify(x_n)

Informal Semantics

A state $\sigma^{\mathcal{I}}$ of a guarded integer program maps each variable to its value. Non-determinism is handled by allowing multiple states to be the outcome of a command. Thus, the execution of a command *cmd* on $\sigma^{\mathcal{I}}$, denoted by $[\![\mathit{cmd}]\!]\sigma^{\mathcal{I}}$, yields a set of states.

Expression evaluation is defined in the standard way. Also, the function $eval(E, \sigma^{\mathcal{I}})$ is the result of the evaluation of the set E of expressions in state $\sigma^{\mathcal{I}}$.

4.3.3 Transformation: C to Guarded-Command

For each C construct we define a transformation to guarded integer commands over the constraint variables. The transformation uses the provided set of contracts and the computed PPT. We define the following transformation functions:

$\boldsymbol{Tr}_{pre}(E)$ is the transformation of the precondition of an expression E.

$\boldsymbol{Tr}(E)$ is the transformation of an expression E.

Next, we elaborate on the transformations of each type of construct. The transformation result of `SkipLine` is displayed in Figure 4.4 and is used in explaining the details of the transformation. For clarity, some of the code is omitted. The transformation uses the contract of `SkipLine` shown in Figure 2.5.

The generated integer program simulates all the possible values which occur in the original program. This is achieved by conservatively interpreting certain statements and by adding non-determinism. In particular, every assignment to constraint variable $b.v$ performs weak-updates in two different cases: (i) b is a summary abstract location representing possibly more than one base address, or (ii) more than one location has the property represented by v.

Technically, we define the following transformation rule

$$b.v \overset{w}{:=} exp \quad \overset{\text{def}}{=} \begin{cases} \begin{array}{l}\text{if } (b.v = \mathit{uninit}) b.v := \mathit{exp} \\ \text{else } b.v := \top\end{array} & count(b) = \infty \\ \\ \begin{array}{l}\text{if } (b.size = \mathit{sizeof}(t) \vee b.v = \mathit{uninit}) b.v := \mathit{exp} \\ \text{else } b.v := \top\end{array} & v \text{ is } \mathit{val} \wedge count(b) \neq \infty \\ \\ b.v := \mathit{exp} & \text{otherwise} \end{cases}$$

In the above rule, summary information is only available and used at transformation time. Strong updates are conducted in 3 cases: (i) the constraint variable is uninitialized, (ii) all locations represented by the abstract base address are overwritten, or (iii) the assignment is to a property of a non-summary

C Construct	Transformation
entry	`assume`($Tr(E_{Pre})$); $\{\text{“}E\text{”}.Pre\} \overset{W}{:=} Tr(E)$; for every expression $\lceil E \rceil_{entry}$ in E_{Post}
`return` E	$\{\texttt{return\_value}_P\} \overset{W}{:=} Tr(E)$;
`exit` P	`assert`($Tr(E\_post(\lceil E \rceil_{entry} / \text{“}E\text{”}.Pre))$);

Table 4.3: The translation of procedure entry and exit points.

abstract location which always has a single value. In our running example, all abstract locations are non-summary. The transformation of statement [1] shown in Figure 4.4 includes a check of weak update.

For ease of understanding, we also define the following sugared construct for set assignments. The syntax

$$X \overset{W}{:=} E$$

where X is a set of variables $x_1, x_2, \cdots, x_n$ and E is a set of expressions $e_1, e_2, \cdots, e_m$ denotes a nondeterministic assignment of one of expressions to one of the variables.

$$\begin{array}{c} x_1 \overset{w}{:=} e_1 \,\Box\, x_1 \overset{w}{:=} e_2 \,\Box\, \cdots \Box\, x_1 \overset{w}{:=} e_m \,\Box \\ x_2 \overset{w}{:=} e_1 \,\Box\, x_2 \overset{w}{:=} e_2 \,\Box\, \cdots \Box\, x_2 \overset{w}{:=} e_m \,\Box \\ \vdots \\ \Box x_n \overset{w}{:=} e_1 \,\Box\, x_n \overset{w}{:=} e_2 \,\Box\, \cdots \Box\, x_n \overset{w}{:=} e_m \end{array}$$

We use general terms in our transformation. However, since the integer analysis can not precisely handle the term $(\neg a) \vee b$, our implementation simplifies such expressions. For example, `assert` $(\neg a \vee b)$ is transformed to `if`(a) `assert`(b) `else skip`. Also, for clarity, when the set is a singleton we omit brackets.

Transformation of Entry and Exit Points

The translation inserts code to reflect procedures' commitments. The translation of procedure P declared as:

$$\begin{array}{lll} t\ P(f_1, f_2, \cdots f_n) & \textbf{requires} & E_{Pre} \\ & \textbf{modifies} & E_{M_1}, E_{M_2}, \ldots, E_{M_n} \\ & \textbf{ensures} & E_{Post}; \end{array}$$

is defined in Table 4.3.

At entry point, the precondition is assumed to hold via an `assume` statement. Additional constraint variables are set to hold the value of expressions needed in the computation of the postcondition. On `return` statement a designated `return_value` variable is set to hold the return value. On exit of procedure P, P's postcondition is verified. In our running example, on entry the `requires` clause is transformed to an `assume` statements. Next, an additional constraint variable is assigned a value which is used in the transformation of the postcondition inserted at the end of the code.

Transformation of C Statements

The transformation includes code for checking the safety of each statement. In addition, the execution of statements may affect properties and values of locations. For this, we need to reflect the effect of assignments to the corresponding constraint variables.

/* void SkipLine(int NbLine, char** PtrEndText) { */
/* requires is_within_bounds(*PtrEndText) && */
    assume($0 \leq rv_{pt}(PtrEndText).\mathit{offset} \leq B.\mathit{size}$);
    assume ($\neg B.\mathit{string} \vee rv_{pt}(PtrEndText).\mathit{offset} \leq B.\mathit{len}$);
/*   *PtrEndText.alloc > NbLine && */
    assume($B.\mathit{size} - rv_{pt}(PtrEndText).\mathit{offset} > lv_{pt}(NbLine).\mathit{val}$);
/*   NbLine >= 0 */
    assume($lv_{pt}(NbLine).\mathit{val} \geq 0$);

    "`*PtrEndText.offset`".$Pre = lv_{pt}(PtrEndText).\mathit{offset}$;
    "`*PtrEndText.alloc`".$Pre = rv_{pt}(PtrEndText).\mathit{size} - lv_{pt}(PtrEndText).\mathit{offset}$;

/* int indice; */
    $lv_{pt}(indice).\mathit{size} := 4$;

⋮

/* [1] indice=0; */
    if ($lv_{pt}(indice).\mathit{size} = 4 \vee lv_{pt}(indice).\mathit{val} = \mathit{uninit}$) $lv_{pt}(indice).\mathit{val} := 0$;
    else $lv_{pt}(indice).\mathit{val} := \top$

⋮

/* [3] if (indice>=NbLine) goto end_loop; */
    if ($lv_{pt}(indice).\mathit{val} \geq lv_{pt}(NbLine).\mathit{val}$) goto end_loop;
    else skip;

⋮

/* [5] *PtrEndLoc = '\n'; */
    assert ($0 \leq lv_{pt}(PtrEndLoc).\mathit{offset} < B.\mathit{size} \wedge$
      $(\neg B.\mathit{string} \vee lv_{pt}(PtrEndLoc).\mathit{offset} \leq B.\mathit{len})$
    if ($B.\mathit{len} = lv_{pt}(PtrEndLoc).\mathit{offset}$)
      $B.\mathit{string} = \mathit{false}$;
    else skip;

/* [6] *PtrEndText = PtrEndLoc + 1; */
    $rv_{pt}(PtrEndText).\mathit{offset} := lv_{pt}(PtrEndLoc).\mathit{offset} + 1$;

⋮

/* [11] *PtrEndLoc = '\0';*/
    assert ($0 \leq lv_{pt}(PtrEndLoc).\mathit{offset} < B.\mathit{size} \wedge$
      $(\neg B.\mathit{string} \vee lv_{pt}(PtrEndLoc).\mathit{offset} \leq B.\mathit{len})$
    $B.\mathit{string} = \mathit{true}$;
    $B.\mathit{len} = lv_{pt}(PtrEndLoc).\mathit{offset}$;
/*} */
/* **ensures** *PtrEndText.is_nullt && */
    assert($B.\mathit{string}$);
/* *PtrEndText.strlen == 0 && */
    assert($B.\mathit{len} - rv_{pt}(PtrEndText).\mathit{offset} = 0$);
/* *PtrEndText.offset == $\lceil$`*PtrEndText.offset`$\rceil_{entry}$ + NbLine */
    assert($rv_{pt}(PtrEndText).\mathit{offset} =$ "`*PtrEndText.offset`".$Pre + lv_{pt}(NbLine).\mathit{val}$);

Figure 4.4: The transformation of `SkipLine`. The original code is commented and the transformed code is indented.

Statement	**Transition**
l: declaration of v of type t	$lv_{pt}(v).SIZE \stackrel{W}{:=} sizeof(t)$
$l : N = (*t)\text{malloc}(E)$ dynamic allocation	$\{b.size \mid \forall b \in allocatedAt(l)\} \stackrel{W}{:=} sizeof(t);$ $lv_{pt}(N).OFFSET \stackrel{W}{:=} 0$
$E_1 = E_2$ any pointer assignment	Assert($Tr_{pre}(E_1)$); Assert($Tr_{pre}(E_2)$); $lv_{pt}(E_1).OFFSET \stackrel{W}{:=} Tr(E_2)$
$*N = E$ integer destructive update	Assert($Tr_{pre}(*N)$); Assert($Tr_{pre}(E)$); $rv_{pt}(N).VAL \stackrel{W}{:=} Tr(E);$ if($Tr(E) = 0$) $rv_{pt}(N).STRING \stackrel{W}{:=} true;$ $rv_{pt}(N).LEN \stackrel{W}{:=} rv_{pt}(N).OFFSET;$ else if ($lv_{pt}(N).OFFSET = rv_{pt}(N).LEN$) $rv_{pt}(N).STRING \stackrel{W}{:=} false;$ else skip;
$E_1 = E_2$ other numeric assignment	Assert($Tr_{pre}(E_1)$); Assert($Tr_{pre}(E_1)$); $lv_{pt}(E_1).VAL \stackrel{W}{:=} Tr(E_2)$

Table 4.4: Transformation of statements given a PPT.

Table 4.4 defines the transformation of basic statements. Procedure calls are discussed separately. On a declaration the transformation assigns to the *size* constraint variable the size of the allocation. Note that the PPT functions $lv_{pt}()$ and $rv_{pt}()$ return sets of abstract locations $\{b_1, b_2, \cdots b_n\}$. We define the function *SIZE* to return the set of $\{b_1.size, b_2.size, \cdots b_n.size\}$ constraint variables.

On a dynamic allocation the $allocatedAt$ mapping of the *PPT* is used to set the size of the abstract location representing base addresses allocated at this statement. This is more precise than setting the size of all abstract locations in the PT-R-value of N. In addition, zero is assigned to the *offset*, indicating that the pointer directly points to the base address. The functions *OFFSET*, $STRING$, *VAL* and LEN are defined in a similar manner to the *SIZE* function.

Destructive updates have the most complicated translation. The translation includes a check for assignment of null termination byte. If so, the assignment to the *string* variable indicates that a null-termination byte exists and the *len* variable is set to the offset of the assigned pointer which is the length of the string. We define the terms $true$ and $false$ as one and zero, respectively. In addition, the case that the null-termination byte is overwritten is handled. The translation of statements [5] and [11] show the two cases of destructive updates. The PT-L-value of `PtrEndLoc` is, as shown in Figure 4.3, the abstract location named B.

Assignments are separated to two cases. Assignments of pointer expressions are translated to reflect the change to the *offset* property. For integer assignments the transformation generate assignments to the *val* constraint variables.

Pointer Exp.		Transformation
$\&v$	simple address-of	0
$\&E.f$	address-of field	$\{e + \mathit{field\_pos}(f) \mid e \in \mathit{Tr}(\&E)\}$
(t) E	pointer-to-pointer casting	$\mathit{Tr}(E)$
(t) E	other casting	$\top$
$E_1 \ \ op_b \ E_2$ where: $\mathit{type}(E_1)$ = t* $\mathit{type}(E_2)$ = `int`	pointer arithmetic	$\mathit{Tr}(E_1) \ \ op_b \ (\mathit{Tr}(E_2) \times \mathit{sizeof}(t))$
E	other	$lv_{pt}(E).\mathit{OFFSET}$

Table 4.5: Transformation of pointer expressions.

Integer Exp.		Transformation
c	constant	c
$E_1 \ \ op_r \ E_2$	boolean expression	$\mathit{Tr}(E_1) \ \ op_r \ \mathit{Tr}(E_2)$
$op_u \ E$	unary operation	$op_u \ \mathit{Tr}(E)$
$E_1 \ \ op_b \ E_2$	binary operation	$\mathit{Tr}(E_1) \ \ op_b \ \mathit{Tr}(E_2)$
(t) E	integer-based casting	$\mathit{Tr}(E)$
(t) E	other casting	$\top$
$*N$	dereference	$(rv_{pt}(N).\mathit{STRING} \ \wedge \ rv_{pt}(N).\mathit{LEN} = lv_{pt}(N).\mathit{OFFSET})?$ $0 : rv_{pt}(N).\mathit{VAL}$
E	other	$rv_{pt}(E).\mathit{VAL}$

Table 4.6: Transformation of integer expressions.

Transformation of C Expressions

As indicated in the translation of statements, the translation of integer expressions and pointer expressions differ. Table 4.5 lists the transformation of pointer expressions. The translation computes the offset property of expressions. The offset of an address-of operation is zero. For field operator the translation uses the *field_pos* information. Casting for one pointer to another does not change the offset of the expression. In a cast from an integer base type to a pointer type the base address is not available. This implies that the offset property cannot be computed. Therefore, we translate it to $\top$ to indicate that all values are possible. For pointer arithmetic, the offset is computed in bytes. Therefore, the size of the type of the pointer operand is used.

The transformation of integer expressions is listed in Table 4.6. The transformation of most expressions is rather straight forward. For dereferences the translation first checks if the dereference value is a null-termination byte.

The transformation of the contract language attributes is defined in Table 4.7. Most attributes are of type integer. The `base` attribute is of pointer type and is transformed to the offset property which is zero for any given base address.

Table 4.8 lists the $\mathit{Tr}_{pre}()$ function which translates the precondition of expressions to integer guarded expressions. This is rather similar to the definition of expressions' precondition in the concrete semantics.

Attribute	**Transformation**
E.base	0
E.offset	$lv_{pt}(E).OFFSET$
E.alloc	$rv_{pt}(E).SIZE - lv_{pt}(E).OFFSET$
E.strlen	$rv_{pt}(E).LEN - lv_{pt}(E).OFFSET$
E.string	$rv_{pt}(E).STRING \wedge rv_{pt}(E).LEN \geq lv_{pt}(E).OFFSET$
overlap(E_1, E_2)	$lv_{pt}(E_2).OFFSET - lv_{pt}(E_1).OFFSET$
is_overlap(E_1, E_2)	$rv_{pt}(E_1) = rv_{pt}(E_2)$

Table 4.7: Transformation of attributes.

Expression	**Precondition**
N reference	$0 \leq lv_{pt}(N).OFFSET \leq rv_{pt}(N).size \ \wedge$ $(rv_{pt}(N).STRING \Rightarrow lv_{pt}(N).offset \leq rv_{pt}(N).len)$
$*N$ dereference where: $type(N) = t*$	$lv_{pt}(N).OFFSET < rv_{pt}(N).SIZE$
$E_1 \ op_b \ E_2$ where: $type(E_1) = t*$ $type(E_2) = \texttt{int}$	$0 \leq (lv_{pt}(N).OFFSET \ op_b \ (tr(E) * sizeof(t'))) < rv_{pt}(N).SIZE \ \wedge$ $(rv_{pt}(N).STRING \Rightarrow (lv_{pt}(N).OFFSET \ op_b \ (tr(E) * sizeof(t'))) \leq$ $rv_{pt}(N).LEN)$

Table 4.8: Transformation of expressions' precondition.

Transformation of Procedure Call

A procedure call statement in the format:

$$N = g(E_1, E_2, \cdots E_n);$$

where g is declared as:

$$\begin{array}{lll} t\ g(f_1, f_2, \cdots f_n) & \textbf{requires} & E_{Pre} \\ & \textbf{modifies} & E_{M_1}, E_{M_2}, \ldots, E_{M_m} \\ & \textbf{ensures} & E_{Post}; \end{array}$$

is transformed to reflect the commitments to the caller and the effect of the call as follows:

Assert($E_{Pre}(f_1/a_1, f_n/a_n, \ldots, f_n/a_n)$);
"E".$Pre :\stackrel{\circ}{=} Tr(E)$; for each $\lceil E \rceil_{entry} \in E_{Post}$
Modify($\{b.v \mid b$ is reachable from $lv_{pt}(E_{M_i}), 1 \geq i \geq m, \forall$ variable $v\}$;
Assume($Tr(E_{Post}(f_i/a_i, \lceil E \rceil_{entry}/$"$E$".$Pre)))$; for each formal f_i, $\lceil E \rceil_{entry} \in E_{Post}$
$lv_{pt}(N) =$ `return_value`;

First, the precondition is verified and values of the current state are stored in the auxiliary variables. The call itself is removed and instead we indicate all representations of locations that may change during the execution as modified. All constraint variables of abstract locations reachable from the expressions specified in the `modifies` clause are set to $\top$. The postcondition is assumed to reflect the commitment of the callee. Finally, the assignment is transformed.

4.3.4 Integer Analysis

An integer analysis is applied to $Tr(P)$, the result of the transformation step to procedure P. This step is similar to the integer analysis step in the SAS algorithm described in Section 3.3.1. The result of this step is a list of potential violations.

When analyzing a nondeterminism statement, we need to take into account all execution options. This is done by analyzing each command and joining the results.

Similarly to the SAS algorithm, a counter example is reported for each potential violation. Since the integer program is over constraint variables of abstract locations in the PPT one may need to view the PPT to understand the error messages.

4.4 Soundness

To prove the soundness of CSSV we need to show that the defined transformation is a correct simulation and that the integer analysis detects all potential string violations. It is strongly based on Lemma 4.2.7 which guarantees that the generated PPT is a sound approximation of the program fragment.

A simulation relation φ between concrete states in the original program and states in the integer guarded program with respect to an α mapping:

$$\alpha\colon \mathcal{L}^\natural(\sigma^\natural) \to BA$$

is defined as:

$$(\sigma^\natural, \sigma^\mathcal{I}) \in \varphi \Leftrightarrow$$

for each $l^\natural \in \mathcal{L}^\natural(\sigma^\natural)$ there exists an abstract location $b = \alpha(l^\natural)$ such that location-simulating formula $\psi(l^\natural, b)$ holds, where

$$
\begin{array}{rll}
\psi(l^\natural, b) \stackrel{\text{def}}{=} & & \text{// size and string properties} \\
& & \mathit{strlen}^\natural(\sigma^\natural)(\mathit{base}^\natural(\sigma^\natural)(l^\natural))) \in \mathit{eval}(b.\mathit{len}, \sigma^I) \wedge \\
& & \mathit{string}^\natural(\sigma^\natural)(\mathit{base}^\natural(\sigma^\natural)(l^\natural))) \in \mathit{eval}(b.\mathit{string}, \sigma^I) \wedge \\
& & \mathit{size}^\natural(\sigma^\natural)(\mathit{base}^\natural(\sigma^\natural)(l^\natural))) \in \mathit{eval}(b.\mathit{size}, \sigma^I) \\
& & \text{// primitive location} \\
& \vee & \mathit{content}^\natural(\sigma^\natural)(l^\natural) \in \mathit{primitive} \Rightarrow \\
& & \mathit{content}^\natural(\sigma^\natural)(l^\natural) \in \mathit{eval}(b.\mathit{val}, \sigma^I) \\
& & \text{// pointer location} \\
& \vee & \mathit{content}^\natural(\sigma^\natural)(l^\natural) \in \mathcal{L}^\natural \Rightarrow \\
& & \mathit{offset}^\natural(\sigma^\natural)(\mathit{content}^\natural(\sigma^\natural)(l^\natural)) \in \mathit{eval}(b.\mathit{offset}, \sigma^I)
\end{array}
\tag{4.1}
$$

We start by claiming that the R-value of an expression is preserved in the simulation.

Lemma 4.4.1 *Given a pair of concrete and integer states $(\sigma^\natural, \sigma^{\mathcal{I}}) \in \varphi$, i.e., $\sigma^{\mathcal{I}}$ is a simulation of $\sigma^\natural$, then for every expression $E \in$* **Expr** *such that $rv(E, \sigma) \notin \{uninit, undef\}$ the following holds:*

$$
\begin{array}{ll}
rv(E, \sigma^\natural) \in \mathit{eval}(\mathit{Tr}(E), \sigma^{\mathcal{I}}) & \textit{where type(E) = integer} \vee \\
rv(E.\mathit{offset}, \sigma^\natural) \in \mathit{eval}(\mathit{Tr}(E), \sigma^{\mathcal{I}}) & \textit{where type(E) = t*}
\end{array}
$$

The proof of this lemma appears in Appendix A.3.

Lemma 4.4.2 *The defined transformation is a simulation with respect to simulation relation φ.*

The proof of the two requirements for a simulation defined in Definition 2.4.1 is :

Base requirement For entry state $\sigma^\natural$ of procedure P exists an entry state $\sigma^{\mathcal{I}} \in \Sigma^\natural_{tr}$ of $\mathit{Tr}(P)$ such that:

$$(\sigma^\natural, \sigma^{\mathcal{I}}) \in \varphi$$

The transformation of an entry is the transformation of the `requires` clause, E_{Pre} as specified in the contract of P. The `requires` clause must hold for an admissible state to arise at entry to P. Thus, $rv(E_{Pre}, \sigma^\natural)$ is true. The initial state, $\sigma^{\mathcal{I}}_0$, in $\mathit{Tr}(P)$ is a single environment in which all variables are mapped to $\top$. Clearly,

$$(\sigma^\natural, \sigma^{\mathcal{I}}_0) \in \varphi$$

The evaluation of $\mathit{Tr}(E_{Pre})$ on $\sigma^{\mathcal{I}}_0$ yields a state $\sigma^{\mathcal{I}}$. Clearly, $\mathit{true} \in \mathit{eval}(\mathit{Tr}(E_{Pre}), \sigma^{\mathcal{I}})$ and since other variables have not changed, we can conclude that:

$$(\sigma^\natural, \sigma^{\mathcal{I}}) \in \varphi$$

Induction requirement Every step preserves the simulation: If $(\sigma^\natural, \sigma^{\mathcal{I}}) \in \varphi$ and $\sigma'^\natural$ is derived by executing statement S on $\sigma^\natural$ then:

$$\exists \sigma'^{\mathcal{I}} \in [\![\mathit{Tr}(S)]\!](\sigma^{\mathcal{I}}) : (\sigma'^\natural, \sigma'^{\mathcal{I}}) \in \varphi$$

The proof is by rule induction on statement S and relies mainly on Lemma 4.4.1 and on the transformation rules. For most statements the transformation rules resemble the semantic definition. Thus, the proof is rather straightforward.

Theorem 4.4.3 *The CSSV algorithm detects all string related errors.*

Proof. We have shown that the transformation is a simulation of procedures. In addition, the transformation creates `assert` commands to verify the cleanness of string manipulations. Since the integer analysis is sound, all potential violations of `assert` statements are detected. ∎

4.5 Empirical Results

Implementing the CSSV tool is non-trivial because of the complicated aspects of C and program analysis. We have implemented a prototype of the CSSV with significant help from the Semantics Based Tools group at Microsoft Research and from Greta Yorsh from Tel-Aviv university. The compiler from C to CoreC is built upon the AST-Toolkit. CSSV uses Golf, a flow-insensitive context-sensitive points-to analysis technique, as the underlying whole program pointer analysis. Golf uses flow edges to represent assignments. Both the integer analysis and the automatic derivation of pre- and post-condition were implemented using the Poly library.

4.6 Benchmarks

We applied CSSV to procedures from the following: (i) A string-manipulation library from EADS Airbus with a total of 400 lines and 11 procedures. (ii) fixwrites — part of web2c a converter from TeX, Metafont, and other related WEB programs to C. `fixwrites` consist of 460 lines and eight procedures. We have manually written contracts for the analyzed procedures.

Table 4.9 describes the benchmark characteristics and the analysis results. The column **LOC** displays the number of source lines in the original source. Column **SLOC** displays the number of source lines after the source-to-source transformation. The **Contract** column investigates the difficulty of manually providing a contract. We use the characters 'S','B' and 'I' as follows: (S) for simple specification, such as `string` and `is_within_bounds`, (B) for specifying the boundaries of buffers, and (I) for other integer relations. There was no need to provide pointer specification in the analyzed code.

Columns **IP Vars** and **IP size** report the number of variables and statements in the integer program produced by C2IP. Columns **CPU** and **Space** display the running time and total allocated space of CSSV. The measurements were done using a 900 MHz Intel Pentium-III CPU with 512MB of memory, running Windows 2000.

4.7 Error Messages

The **Msg** columns classify the messages reported by CSSV. Messages are classified as errors for cases where there is an input to the application on which the error occurs. The errors detected are due to unsafe calls to library functions, such as `strcpy()`, unsafe assumptions that an input contains a specific character, or unsafe pointer arithmetic.

CSSV verifies the absence of errors on most procedures of Airbus. There are six false messages on a single procedure of Airbus. The program destructively assigns a non-zero character to a certain place in a buffer. CSSV fails to infer that this character is non zero. The function `skip_balanced` safely assumes that the input parameter contains a balanced number of parentheses. This is verified by the `whole` function which is called prior to `skip_balaned`. This example demonstrates that in some cases it is hard to separate safety from correctness. To show that this function is safe, we need to verify correctness, i.e., that the implementation correctly checks that the input string contains a balanced number of parentheses. Fortunately, in most of the analyzed examples, this is not the case, i.e., the safety does not depend on correctness.

4.8 Conclusion

Buffer overflow is one of the most harmful source of defects in C programs. Moreover, it makes software vulnerable to hacker attacks. CSSV demonstrates that buffer-overflow defects in software can be

App.	Function	Source Code			CSSV				Msg	
		LOC	SLOC	Contract	IP Vars	IP Size	CPU sec	Space MB	False	Errors
EADS Airbus	RTC_Si_SkipLine	13	260	SBI	39	109	2.6	12	0	0
	RTC_Se_CopieEtFiltre	66	773	SB	127	812	206	347	6	0
	RTC_Si_FiltrerCarNonImp	19	114	S	13	151	0.3	2	0	0
	RTC_Si_Find	26	820	SI	108	476	2.7	24	0	0
	RTC_Si_StrNCat	8	299	SBI	54	182	0.9	6	0	0
	RTC_Si_CalculerStringTime	33	567	SBI	86	529	76	127	0	0
	RTC_Si_FormatMcduTo-Formatprinter	18	273	SB	58	323	6.9	28	0	0
	RTC_Si_StoreIntInBuffer	35	222	SBI	59	346	9.8	43	0	0
	RTC_Se_ComposerEntete	10	550	SI	77	352	3.4	23	0	0
fixwrites	remove_newline	12	260	S	35	203	0.1	2	0	0
	insert_long	14	367	SB	138	571	13	99	0	2
	join	15	701	SB	95	443	2.1	23	0	2
	whole	30	423	S	46	352	1.2	20	0	1
	skip_balanced	20	258	SB	29	215	0.3	5	2	0
	bare	26	333	S	41	319	0.6	12	0	3

Table 4.9: The experimental results.

detected with a modest burden on the software developer by way of a reasonable effort to add contracts. In other words, CSSV verifies the absence of buffer overflow in complex C procedures with few false alarms.

Usability of the whole system is improved by eliminating the need for contacts regarding pointer values which are, in many cases, not part of the "logic" of the procedures. We have shown how to construct a procedural points-to information that is scalable — due to the use of a global flow-insensitive pointer analysis, and rather precise — due to the novel algorithm for extracting a procedural points-to information from the global one. We believe that CSSV provides evidence that sound analysis can be applied to verify statically the absence of all string errors in realistic applications.

Chapter 5

A Comparison Between the SAS and CSSV Algorithms

The SAS and CSSV algorithms detect potential string violations. As indicated before, CSSV can handle all C constructs including multilevel pointers and does not depend on the correctness of the contracts. In addition, CSSV is much more efficient. However, in some cases the SAS algorithm is more precise than the CSSV algorithm. This chapter shortly explains this issue.

The underling analysis of CSSV is a flow-insensitive pointer analysis technique. Although the SAS algorithm does not compute points-to information, the overlapping information contains points-to relationships. The overlapping information is flow-sensitive.

For example, consider the code in Figure 5.1. The CSSV algorithm reports a false message concerning the fisrt `strcpy` call statement. This stems from the fact that the pointer analysis is flow-insensitive. CSSV takes into account that `p` may point to `short` which is ten bytes long and therefore the precondition of `strcpy` may not hold.

However, the SAS algorithm which computes overlapping determines that on the first call to `strcpy` `p` definitely overlaps `big` by zero bytes. Thus, `p` points to a one-hundred byte long buffer and the call is safe. This is because the overlapping information is updated in a flow sensitive manner.

```
void main()
{
  char big[100];
  char small[10];
  char *p;

  if ( ··· ) {
     p = big;
     strcpy(p, "This is a long string");
     }
  else {
        p = small;
        strcpy(p, "short");
        }
  }
```

Figure 5.1: Example of a program for which CSSV produces a false alarm while SAS shows the absence of errors.

Chapter 6

CLLV: C Linked List Verifier

Many nasty bugs result from memory cleanness errors such as `NULL` pointer dereferences. These errors should never occur in bug-free programs, regardless of the specification.

In this chapter we describe algorithms that automatically uncover memory cleanness errors in C programs. Following the theme of this thesis, the algorithms are conservative. They never miss an error but they may generate false alarms. The algorithms detect the following types of memory cleanness errors:

- NULL dereference
- Memory leakage
- Usage of uninitialized and dead storage

Clearly, to detect memory errors in a language like C, one needs to reason about the pointer behavior of the program. However, this work does not intend to introduce new shape analysis or pointer analysis algorithms. Instead, we focus on detecting cleanness violations by using existing algorithms. We applied the algorithms to C programs which manipulate singly linked lists. Two major results have been obtained:

- A new, efficient and quite precise algorithm for detecting memory cleanness errors (see Section 6.1). It does not yield **any** false alarm on quite intricate C programs (see Table 6.1). On programs manipulating more complicated data structures such as cyclic lists and tree, it produces false alarms[1]. It represents memory states as Shape Graphs with Reachability from stack variables (see [18]) and thus we call it *SG+R*. To assess the usability of our new algorithm we implemented it using PAG – a program analyzer generator developed at Saarland University [46]. On bad examples *SG+R* can be rather slow. However, it is reasonably efficient on non-trivial programs that we tried. It runs significantly faster than the algorithm described in [18] since it avoids the cost of transitive closure. It also compares favorably with verification systems aimed at showing memory cleanness since it does not require loop invariants. Interestingly, *SG+R* runs significantly faster than the verification system described in [38] on the programs we tried. For example, on the procedure `swap` shown in Figure 6.1(c) the *SG+R* algorithm runs in less than 0.01 seconds as opposed to 21.
- We use two of the well known pointer analysis techniques to check cleanness and compare their cost and effectiveness to our new algorithm. The first technique is **flow-sensitive points-to** analysis [2, 62] (referred to as *PT*). The second algorithm, *SG*, uses Shape Graphs and is a slight

[1] Further work reported in [55] avoids these problems

program	description
search.c	searches for an element in a linked list
null_deref.c	searches a linked list but with a typical error of not checking for the end of the list
delete.c	deletes a given element from a linked list
del_all.c	deletes an entire linked list
insert.c	inserts an element into a sorted linked list
merge.c	merges two sorted linked lists into one sorted list
reverse.c	reverses a linked list via destructive updates
fumble.c	an erroneous version of reverse.c which loses the list
rotate.c	performs a cyclic rotation when given pointers to the first and last elements
swap.c	swaps the first and second elements of a list, fails when the list is one element long (see Figure 6.1)

Table 6.1: Description of the analyzed programs. Some of these interesting programs are from *LCLint*[25], [38] and from first-year students. They are all in the format shown in Figure 6.1 and available at http://www.math.tau.ac.il/∼nurr.

variant of the shape analysis technique described in [58]. The *SG* algorithm is similar to *SG+R* except that it does not (explicitly) track reachability from stack variables. Our results are reported in Table 6.2 and Table 6.3. We compare time, space and the number of false alarms. We believe that space is more important than time since it is less implementation dependent and since it is normally the bottleneck in scaling program analysis algorithms to handle large programs (see also Section 6.2). We conclude that:

- As expected, the *PT* algorithm is the cheapest of the three but it is quite imprecise. In fact, it cannot be used at all to check for memory leakage errors which are quite problematic.
- The *SG* algorithm is less effective than the *SG+R* algorithm and is sometimes slower. This is evidence to a general phenomenon in program analysis where a less precise algorithm can be sometimes less efficient in practice since the less precise algorithm may fail to detect the important program invariants, thereby, abstracting situations that can never occur in the program and thus consuming more time and space.

The *SG+R* algorithm does not handle pointer arithmetic, casting, and function pointers. There are also a few simplifying assumptions for ease of implementation. These features, which we can handle but prefer to ignore in this implementation, are: (i) address-of; (ii) arrays; and (iii) recursive functions. The current implementation analyzes programs and handles procedures by call-string [63] which is provided by PAG.

The rest of this chapter in organized as follows. Section 6.1 describes the *SG+R* algorithm. Section 6.2 describes the implementations and compares the algorithms. We conclude by assessing the potential and limitations of our approach (Section 6.3).

6.1 The *SG+R* Algorithm

This section describes the *SG+R* algorithm for memory cleanness checking. The algorithm is a standard iterative dataflow analysis algorithm (e.g., see [48]). It iteratively computes a finite set of “abstract

program	*SG+R*		*SG*		*PT*	
	time	space	time	space	time	space
search.c	0.02	860	0.04	934	0.01	357
null_deref.c	0.03	968	0.02	1042	0.02	403
delete.c	0.05	1632	0.1	2147	0.01	560
del_all.c	0.02	400	0.01	446	0.01	317
insert.c	0.02	1834	0.05	2272	0.03	1035
merge.c	2.08	4931	1718	138466	0	810
reverse.c	0.03	922	0.03	946	0.01	380
fumble.c	0.04	868	0.04	803	0.02	343
rotate.c	0.01	886	0.01	969	0	552
swap.c	0	478	0.01	573	0	295

Table 6.2: The CPU time (in seconds) and space (counted as the total number of edges, see Section 6.2) used to analyze the tested programs using the three implementations.

program	real errors	false alarms					
		SG+R		*SG*		*PT*	
		leak	ref	leak	ref	leak	ref
search.c	0	0	0	2	0	5	0
null_deref.c	3	0	0	2	0	5	0
delete.c	0	0	0	2	0	7	0
del_all.c	0	0	0	0	0	6	4
insert.c	0	0	0	2	0	7	1
merge.c	0	0	0	6	0	8	5
reverse.c	0	0	0	0	0	7	0
fumble.c	1	0	0	0	0	6	0
rotate.c	0	0	0	2	0	5	1
swap.c	3	0	0	0	0	5	0

Table 6.3: The precision of the three algorithms in terms of false alarms reported on the tested programs (see Section 6.2).

descriptions" of all the stores that occur at a given program point. It starts with the abstract description of the initial store at the beginning of the program and then it repeatedly applies *transfer functions* for every program statement and condition until a fixed point is reached. Upon termination, it checks the cleanness conditions against the abstract description.

The algorithm is conservative, i.e., upon termination, every store that can possibly occur when the execution of a program reaches a given program point, pt, is described by some abstract description computed at pt. However, it may happen that an abstract description at pt describes a store that cannot occur at pt, either because the class of inputs that is being considered is too broad or since the algorithm over approximates the storage configurations. This means that the algorithm can never miss a potential error but may report false alarms.

The rest of this section is organized as follows: In Section 6.1.1, our memory abstraction is described. Then, in Section 6.1.2, we demonstrate the transfer functions. Finally, in Section 6.1.3 we explain how cleanness is checked.

6.1.1 Shape Graphs with Reachability

A *shape graph* is an abstraction of the memory states in which the actual values of both data and pointer cells are ignored. Instead, the shape graph captures the "shape" of the data structures and the relationships between different stack (and static) pointer variables. Heap cells are represented by *shape-nodes* and sets of "indistinguishable" heap cells are represented by a single shape-node, called a *summary-node*. The number of these shape graphs depends on the program but not on its input.

Figure 6.1(a) shows a declaration of a linked list data type in C and the main function that creates the list and then invokes the `swap` function shown in Figure 6.1(c) that swaps the first two elements via destructive updating. This program is used as a running example in this section. Figure 6.2 shows the three shape graphs that occur when the *SG+R* algorithm analyzes the `create_list` function: the shape graph g_1 represents the situation in which `c` is `NULL`; g_2 represents a one element linked list, g_3 represents linked list of two or more elements pointed-to by `c`. In g_3, the shape node n_2 represents the tail of the list pointed-to by `c`.

Formally, a shape graph is a directed graph with a set of shape-nodes, *variable-edges* connecting pointer variables to shape nodes, and *selector-edges* connecting shape nodes.

Shape Node

A shape node n has the following properties:

- The set $ptb(n)$ consists of the stack variables that are directly pointing to all the concrete heap cells represented by n. In g_3, $ptb(n_1) = \{c\}$ and $ptb(n_2) = \emptyset$. In the shape graph there is a variable-edge from every variable in $ptb(n)$ to the node n. For clarity, the ptb property is not displayed in the figures inside the shape node. Note, that exactly one heap cell is represented by n if $ptb(n)$ is not empty.

- The set $reach(n)$ consist of the stack variables that via (chains of) selectors reach all the concrete heap cells represented by n. In graph, g_3, $reach(n_1) = \{c\}$ and $reach(n_2) = \{c\}$ since all heap cells represented by n_2 are reachable-from c. Pictorially we display the variables in $reach$ of a node below the name of the node.

  The shape graph shown in Figure 6.3 corresponds to a memory state in which `c` points to a linked list of length four or more, and `x` points in between the second and the last element of the list. Node n_1 represents the first list element, n_3 represents the element pointed-to by `x`. Node n_2 represents all the elements between the element pointed-to by `c` and the element pointed-to by `x`.

```
typedef struct
node {
  struct node *n;
  int data;
} List;


main() {
  List *c;
  c = create_list();
  c = swap(c);
}
```

(a)

```
List* create_list() {
  List *e,*c;
  int i,size;

  c = NULL;
  scanf("%d",&size);
  for(i=0; i<size; i++) {
     e = malloc(sizeof(List));
     e->data = i;
     e->next = c;
     c = e;
     }
  return c;
}
```

(b)

```
List* swap(List *c) {
  List *p;
  s1: if ( c!=NULL) {
  s2:    p = c;
  s3:    c = c->n;
  s4:    p->n = c->n;
  s5:    c->n = p;
      }
  s6: return c;
}
```

(c)

Figure 6.1: A C program with (a) a declaration of a linked list data type and a main function, (b) a function that creates a `NULL` terminated list and (c) a function that uses destructive updating to swap the first two elements of a list.

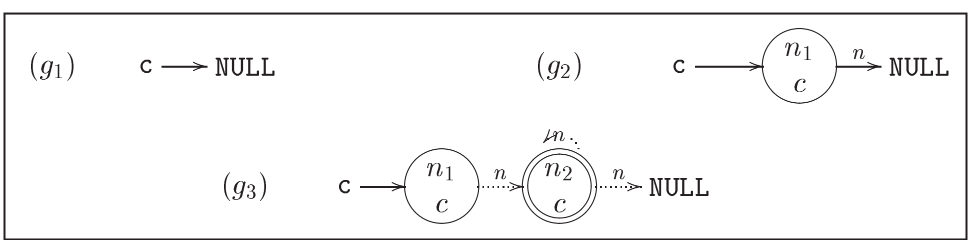

Figure 6.2: The shape graphs that arise from the analysis of the `create_list` function shown in Figure 6.1(b). Shape nodes are circles, named by n_i. Appearance of name p below the name n_i means, "the cell(s) represented by n_i are reachable from p", i.e., $p \in reach(n)$.

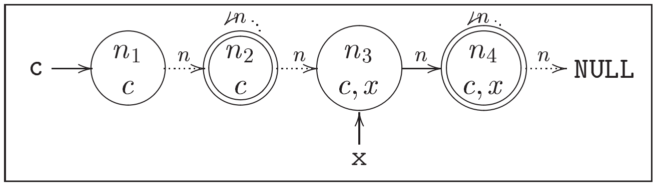

Figure 6.3: A shape graph that arises in the *SG+R* algorithm when x traverses a linked list pointed-to by c.

It has the properties $ptb(n_2) = \emptyset$ and $reach(n_2) = \{c\}$. Node n_4 represents all the list elements that are the tail of the list, it has the properties $ptb(n_4) = \emptyset$ and $reach(n_4) = \{c, x\}$. Nodes n_2 and n_4 are distinguishable as they represent disjoint sets of heap cells.

- The Boolean flag $unshared(n)$ holds the value $true$ for nodes representing heap cells that cannot be pointed-to by more than one selector. In g_3, $unshared(n_1) = true$ and $unshared(n_2) = true$, since no heap cell represented by n_2 can be pointed-to by two or more selectors. This implies that two incoming selector-edges into n_2 cannot simultaneously occur in any concrete state of data store.

  As a matter of notation we use the term $is(n)$ to describe nodes n with $unshared(n) = false$. For example, n_3 in shape graph g_{12} of Figure 6.4 is shared.

- The Boolean flag $alloc(n)$ indicates nodes which represent allocated heap cells. This flag is false for nodes representing freed memory cells. We need this information to detect usage of dead storage.

- The Boolean flag $unique(n)$ indicates that n is a *unique node*, i.e., it represents exactly one heap cell. Nodes n with $unique(n) = false$ may represent more than one heap cell and are therefore called *summary node*s. The node n_2 in g_3 is a summary node. Summary nodes are drawn as double framed circles.

There is a special shape node NULL representing NULL pointers. It is important to understand that properties of a node are conservative approximations of the actual properties of all the heap cells represented by that node. For instance, every pointer variable in $reach(n)$ must reach all heap cells represented by n but the converse need not hold, it may be that a variable x does not belong to $reach(n)$ and yet for every heap cell, l, represented by n there is a selector path from x to l. In particular, a node n with $reach(n) = \emptyset$ may represent non-garbage storage. A similar situation holds for the Boolean flags. If some Boolean property, B, holds for a node n then for all heap cells represented by n B holds. The opposite need not hold. For example node n with $unique(n) = false$ may represent a single heap cell.

Selector Edges

There are two types of selector-edges: *definite* and *indefinite*. Definite edges are drawn as solid lines and indefinite edges are drawn as dotted lines (the reader may notice the analogy with [59] which is not explained here for reasons of space). Selector edges are labeled with the name of the selector.

An indefinite selector-edge from a node m into n indicates that the heap cells represented by m may have a selector into a heap cell represented by n. In g_3, the indefinite edge from n_2 to itself represents selectors into next list elements. This is an indefinite edge since it might not hold for all the heap cells represented by n_2, e.g., between the second and the fourth element. In some situations, the analysis can be more precise by taking into account the fact that a selector-edge is definite, i.e., it always exists.

6.1.2 Handling Statements and Conditions

In this section we sketch the application of the iterative algorithm to the running example. The algorithm iteratively computes a set of shape graphs at every program point. To guarantee that the number of shape graphs is finite, we merge shape nodes with the same properties to summary nodes. This causes a loss in precision but ensures that the algorithm terminates. Note that the number of shape nodes can be exponential in the number of stack variables. However, measurements reported in the next section seem to indicate that this is not a problem, at least for the tested programs.

The iterative algorithm analyzes the running example by applying the sequence of transfer functions belonging to the statements of the swap function to each of the shape graphs in Figure 6.2. Conditions are partly analyzed and shape graphs are not propagated if the condition is not met. In the analysis of g_1 the condition at statement s_1 is not met and the shape graph is immediately propagated to the `return` statement s_6. The analysis of g_2 is not explained here. Instead, Figure 6.4 follows the evolution of g_3, which is the interesting one. The condition at statement s_1 is met and g_3 is propagated to statement s_2.

Statement s_2 generates a shape graph representing concrete stores where `c` and `p` point to the heap cell represented by n_1. Therefore, in g_5, node n_1 represents a heap cell pointed-to by `c` and `p` and thus $ptb(n_1) = \{c, p\}$. Also, since n_2 represents heap cells reachable-from `c` before the statement, after the statement it represents heap cells reachable-from `c` and `p` and thus $reach(n_2) = \{c, p\}$. We see that for this statement, the update of reachability information is fairly simple. We defer the explanation of updating reachability of general assignments to the end of this subsection.

One of the interesting and complicated aspects of shape analysis is the way linked lists are traversed. Statement s_3 demonstrates this: in the shape graph before this statement (g_5) we know that n_2 is unshared ($unshared(n_2) = true$). Therefore (only) two types of concrete memory states can be represented by this shape graph: (i) memory states in which `c` and `p` both point to lists with exactly two elements, and (ii) memory states in which `c` and `p` both point to lists of length three or more. The shape analysis algorithm conservatively assumes that both situations can occur. Hence, the statement `c=c→n` leads to two shape graphs g_6 and g_7. The difference between these graphs is the presence of a summary node n_3 in g_7 indicating that the list contains more than two elements. In both of these shape graphs, n_1 represents heap cells pointed-to by (and reached-by) `p` while n_2 represents heap cells pointed-to by `c` and reachable-from `p` and `c`. Node n_3 in g_7 represents heap cells not pointed-to by any variables but reachable-from `p` and `c`. This operation of advancing a pointer down the list is called *materialization* in [7, 58] since a new shape node (in this example the node pointed-to by `c` was materialized).

Now, statement s_4 is analyzed on the two shape graphs resulting from statement s_3. Our implementation uses a front-end to break complicated statements into simpler ones in a usual way. Every assignment of the form $l = r$ is performed by: (i) generating a new temporary, say t, to store r; (ii) uninitializing l, denoted by `kill(`l`)`; (iii) assigning t to l; and (iv) uninitializing t (of course the first and the last stages can be sometimes avoided). So far, we have ignored this issue for ease of understanding. Since s_4 is rather complex, we chose to break this statement into four stages (as in the implementation). For reasons of space, we show two stages in each row in Figure 6.4.

The effect of statement `t = c→n` on g_6 is simply to assign `NULL` to `t`, and then `kill(p→n)` removes the selector edge from the heap cell pointed-to by `p`. As a result, in g_8, n_1 does not have selector edges. Also, the only heap cell reachable-from `p` is the one directly pointed-to by `p`, represented by n_1.

The effect of statement `t = c→n` on g_7 is to materialize a new node representing the heap cell pointed-to by `t`, thus generating two graphs and then `kill(p→n)` removes the selector edge from the heap cell pointed-to by `p`. As a result, node n_1 in the resulting shape graphs g_9 and g_{10} does not have selector edges. Also, the only heap cell reachable-from `p` is the one directly pointed-to by `p`, represented by n_1.

Next, the assignment `p→n = t` is performed on g_8 which connects the selector edge of n_1 to `NULL`

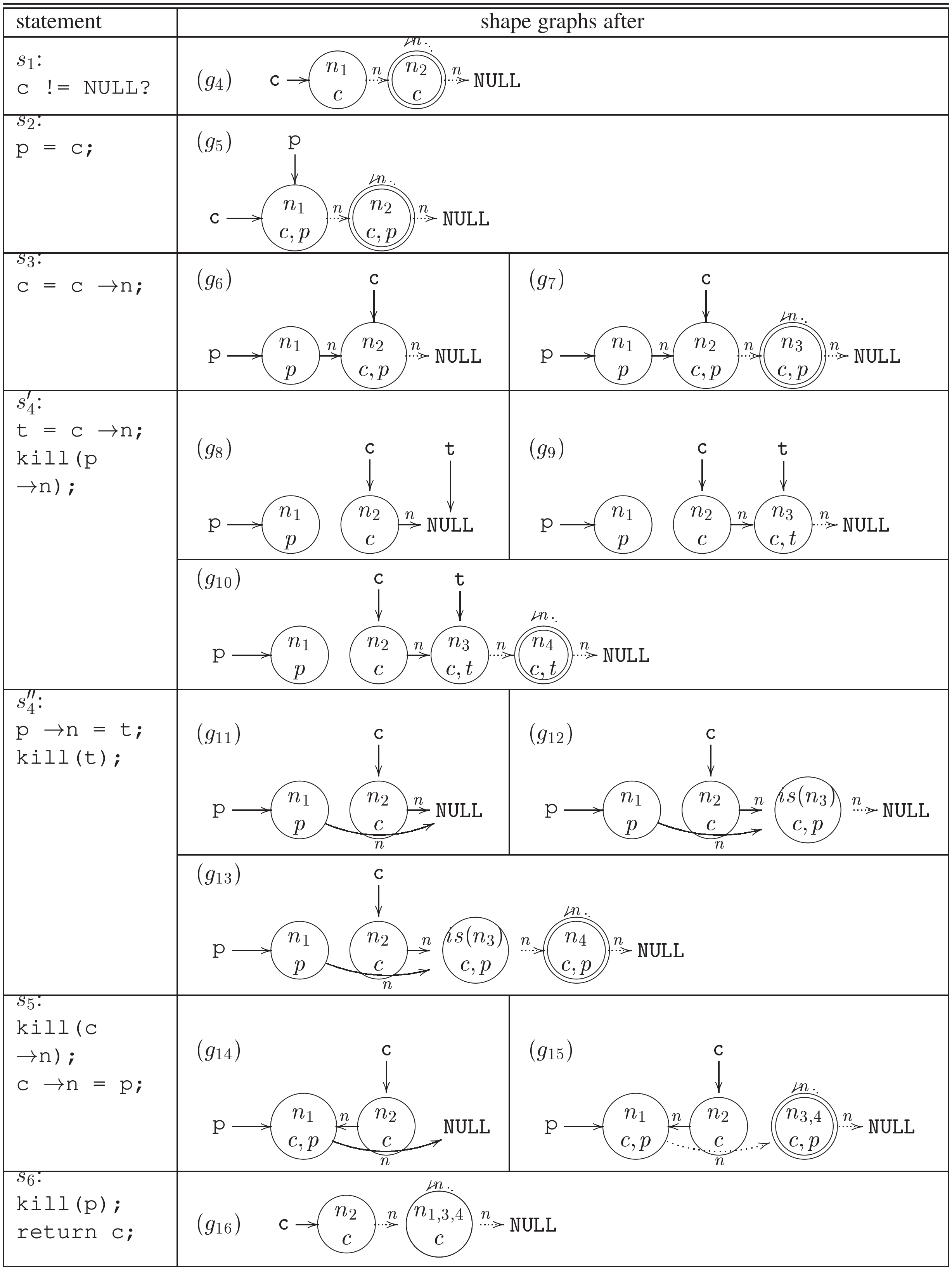

Figure 6.4: The shape graphs that arise when the *SG+R* algorithm is applied to the function in Figure 6.1(c) with the input shape graph g_3 of Figure 6.2.

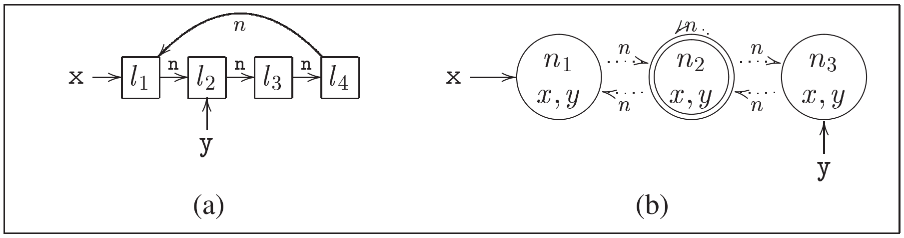

Figure 6.5: A cyclic concrete store (a) and the corresponding shape graph (b).

and then `kill(t)` results in g_{11}. Performing this assignment on g_9 and g_{10} is more interesting since before this statement the heap cell represented by n_3 may have an incoming selector edge. Therefore, this statement results in $unshared(n_3)$ set to $false$. After `kill(t)`, we get the graph g_{12} (from g_9) and g_{13} (from g_{10}). Note that n_3 and n_4 in g_{13} are not merged since n_3 has $unshared = false$ and n_4 has $unshared = true$. As we shall see, this is what allows us to eventually conclude that `c` points to a linked list at s_6. The update of reachability for this statement is explained in Section 6.1.2.

The statement `kill(c→n)` at s_5 also performs destructive update on the selector from `c` which removes the selector edge from the heap cell pointed-to by `c` into the heap cell pointed-to by `c→n`. Therefore, the analysis removes the selector edge from n_2. Moreover, the heap cell represented by n_3 is unshared, causing $unshared(n_3)$ to be set to $true$, but now n_3 and n_4 have the same properties (excluding $unique$ which is not used as a node distinction). Thus, these nodes are merged, and therefore after `c→n = p` we get g_{15} where node $n_{3,4}$ corresponds to n_3 and n_4 in g_{13} and to n_3 in g_{12}.

Before statement s_6, we join the abstract representation from the true path and the false path of the `if` statement at s_1. Since at each program point, we maintain a set of shape graphs, the join is a set-union operation generating the set containing graphs g_1,g_{14},g_{15} and more (those evolved from g_2). This operation is not shown in Figure 6.4 since it only follows the evolution of g_3.

At statement s_6, the local variables are uninitialized and thus `kill(p)` causes $ptb(n_1)$ from g_{14} to be set to empty, and thus nodes n_1 and $n_{3,4}$ are now merged, resulting in g_{16}.

The reader is referred to [58, 59, 52] for more elaborations on shape analysis algorithms. Our way of conservatively and efficiently updating reachability in linear time is new and thus explained in the next subsection.

Updating Reachability

A statement such as s_4 which assigns a new value to a selector of a variable `p` can affect the reachability from other stack variables "upstream" from `p→n`. Figure 6.5(a) contains an interesting concrete store which demonstrates the most complicated case arising in cyclic graphs. If the statement `kill(y→n)` is performed on this store, l_4 will no longer be reachable-from `x` but l_2 (and l_1 and l_3) will. Our current analysis therefore conservatively handles reachability in this case: we remove the reachability from all the variables `x` that can reach the heap cell pointed-to by `y`. Thus, when the statement is performed on Figure 6.5(b), we conservatively lose the fact that x is in $reach(n_2)$. In our running example as well as in all the other programs this does not lead to false alarms since no cyclic lists are created. The advantage of our simplified approach is that it can be implemented in time linear in the size of the shape graph. A more precise and more expensive solution is described in [57].

6.1.3 Cleanness Checking

Every statement has a cleanness precondition, i.e., a requirement that every store occurring at this statement must satisfy. For example a statement, st, of the form `y = x` requires that `x` points to either `NULL` or to an allocated heap cell before this statement. For example, st is not clean when `x` is uninitialized or refers to a freed heap cell. Our tool conservatively checks cleanness preconditions by investigating the resulting shape graphs. For each shape graph g that occurs before a statement st, we impose conditions that guarantee that the preconditions of st are met by every store represented by g. These conditions are explained below. They can be justified using the theory of abstract interpretation, see [11]. This is beyond the scope of this paper. Since the iterative algorithm is itself conservative, we conclude that our tool can never miss a cleanness violation.

For each program point, our tool displays the shape graphs that violate the cleanness conditions with an appropriate error message. The programmer can see results of the analysis graphically and determine whether it is an error or a false alarm. Of course, false alarms can be generated when the shape graph at a given point describes memory states that can never occur at that point. However, since the analysis is conservative, it will never miss a cleanness violation. We try to avoid reporting cleanness errors from the same cause at different program points.

We conclude this subsection by specifying the cleanness conditions on shape graphs.

Pointer Reference Condition

When referring to a pointer `x` we check that there is a definite variable-edge from x to a node n with $alloc(n) = true$. (For simplicity, the `NULL` node has $alloc = true$). Here, we use the fact that the analysis guarantees that if `x` can be uninitialized it must yield a shape graph where there is no variable-edge from x. Similarly, if `x` may point to a freed storage, there must be a shape graph where there is a selector edge from x to a node n with $alloc(n) = false$. All references in our running example are safe.

Pointer Dereference Condition

Every pointer dereference should produce a reference and therefore the pointer reference condition must hold and also there should be no variable edge from x to the `NULL` node. For example, when analyzing the `swap` function in Figure 6.1(c) on the shape graph g_2 in Figure 6.2 we report a potential dereference to `NULL`. Figure 6.6 shows the shape graph that arises. There is a variable edge from c to the `NULL` node. Therefore, the statement `p→n=c→n` is not safe. The tool reports an error message with this shape graph.

Memory Leakage Condition

The memory leakage precondition is more complicated and we will explain it on each of the two operations `kill(x)` and `kill(x→n)`, where memory leakage can occur. We only check memory leakage if the statement does not raise other cleanness errors explained before.

When the statement `kill(x)` is applied, we need to assure that either: (i) `x` is uninitialized; (ii) `x` is pointing to `NULL`; or (iii) `x` is pointing to a heap cell which is also reachable from a different stack variable `y`. On the shape graph we check that either: (i) There is no variable edge from x; (ii) There is a variable edge from x to `NULL`; or (iii) There is a variable edge from x to a node n and either: (a) there is a different variable `y`, $y \in ptb(n)$ or $y \in reach(n)$; or (b) there is a definite incoming selector edge into n.

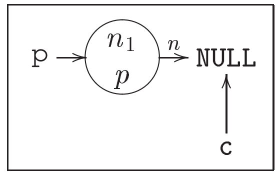

Figure 6.6: A shape graph that arises after statement s_3, `c=c→n`, when the shape analysis algorithm is applied to the function in Figure 6.1(c) with input shape graph g_2 in Figure 6.2.

The condition (iii)(b) may seem surprising since a definite incoming selector-edge may only emanate from a node n' representing garbage heap cells. In this case we choose not to report memory leakage since this error was issued when n' became inaccessible.

When the statement `kill(x→n)` is applied, we need to assure that either: (i) `x→n` is uninitialized; (ii) `x→n` is pointing to `NULL`; or (iii) the heap cell pointed-to by `x→n` can be reached from a different stack variable `y`.

On the shape graph, let n_x be target of the variable-edge from x (which must exist since this statement does not raise a dereference violation). We check that either: (i) There is no selector-edge from n_x. (ii) There is a unique selector-edge from n_x and it is to `NULL`. (iii) For each successor node, n', of n_x (a node that has an incoming selector-edge from n_x) either: (a) there is a different variable `y`, $y \in ptb(n')$ or $y \in reach(n')$, or (b) there is a definite incoming selector edge into n'.

Note that this case is slightly more complicated than the `kill(x)` case mainly because there can be more than one outgoing selector-edge from n_x. As before we try to avoid reporting the same error twice.

6.2 Comparison

In this section we describe the implementation of the algorithms and analyze the results of applying the algorithms to the programs listed in Table 6.1.

6.2.1 Implementation Issues

All algorithms were implemented using PAG. The time is measured by PAG running on a Linux system on a PC Pentium 166 Mhz with 128MB.

All analyses conservatively take into account conditions of the form of pointer (in)equalities. For ease of implementation, the treatment of `NULL` pointers is improved by source-to-source transformation that uses a special variable to indicate `NULL`. The cleanness checking is for: (i) `NULL` dereference; (ii) usage of uninitialized pointers; (iii) usage of dead storage; and (iv) memory leakage.

The *PT* analysis computes the well-known points-to information, [2, 62]. It represents the possibly many objects allocated at calls to `malloc` by creating a named object based on the number of the node in the control flow graph (CFG). The analysis is flow-sensitive, i.e., it considers the control flow of the program. For detecting usages of uninitialized storage we keep a special location called `garbage` which all pointers are initialized to. An additional analysis that enables the removal of over-conservative selector edges to garbage is carried out. The analysis indicates storage that may have been freed.

Cleanness checking in the *PT* algorithm is rather straightforward. The cleanness preconditions are: (i) In a reference to pointer `x`, `x` should not point to `garbage` or to a possibly freed storage; (ii) In a dereference to pointer `x`, `x` must not point to `NULL`; (iii) In assignments to `x`, the precondition is that another pointer or selector points to the same location as `x`. Note that in our example the pointers point to the heap. Therefore, in every assignment to a previously assigned pointer we report a potential memory leakage. This stems from the fact that we can not guarantee that another pointer points to the same location as `x`.

The *SG* algorithm is similar to the *SG+R* algorithm but it does not track down the reachability information. A node in the shape graph is not characterized by the set of reached-by variables. In practice, the *SG* and the *SG+R* algorithms are the same implementation with a flag indicating whether or not to update the reachability. Cleanness checking in the *SG* algorithm is exactly the same as in the *SG+R* algorithm but in the *SG* algorithm $reach(n)$ is always empty.

The *SG* algorithm is a slight improvement, in terms of precision of the [58] algorithm in that the *SG* algorithm computes a set of shape graphs for each point in the program while the [58] algorithm combines the different shape graphs at a program point to one shape graph. This implies that the [58] algorithm will have at least as many false alarms as the *SG* algorithm but we do not know whether it will run faster.

Precision

Table 6.3 lists the number of real errors in the program (three of the tested programs are incorrect) and the number of false alarms reported by the algorithms. All algorithms are conservative, they do not miss any errors. Therefore, the interesting number is the number of false alarms. Measuring false alarms provides a true indication as to the precision of the pointer analysis as opposed to, say, the number of shape graphs or points-to pairs. We classify the false alarms into two types. The first type, *leak*, are false alarms on memory leakage. The *ref* false alarms column counts pointer reference and pointer dereference false alarms (see Sections 6.1.3 for the *SG+R* algorithm). The reason for this separation is that one does expect the *PT* algorithm to effectively detect pointer reference errors but not memory leakage errors.

From the table we can establish the following observations:

- The *SG+R* algorithm yields no false alarms on the tested programs. This indicates that the abstraction used by the *SG+R* algorithm has the exact precision needed for analyzing the linked list programs. It does generate false alarms on tree manipulation programs or when infeasible control flow paths are conservatively taken into account.

- The *SG* algorithm does not yield false alarms of type *ref*. This indicates that the relation between program variables which is kept by the abstraction of the *SG* algorithm in the nodes' ptb property is enough to avoid *ref* false alarms.

- Explicit reachability information is needed to avoid false alarms on memory leakage. The *SG* algorithm reports false alarms on memory leakage on almost all programs. A common false alarm, repeated in many programs, is reported upon correctly traversing a list. Figure 6.7 is an example of a shape graph that arises when traversing a list inside a loop. This graph represents a data structure where `c` points to the head and `x` points to an element (third and on) inside the structure[2]. Processing the statement `x=x→n` generates a false alarm on memory leakage. This is because node n_3 is not pointed-to by any other variable, it does not have a definite incoming edge and we do not know that it is reachable-from c. In the *SG+R* algorithm these false alarms are avoided since we know that the node is reachable-from c.

- The *PT* algorithm is useless for checking memory leakage. Every assignment to a previously allocated heap pointer is a potential leakage. This is unfortunate since there are true bugs in

[2]From this shape graph we do not know that it represents a list.

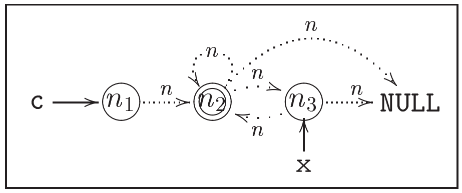

Figure 6.7: A shape graph that arise by the *SG* algorithm when correctly traversing a linked list. Processing the statement x=x→n on this shape graph yields a false alarm.

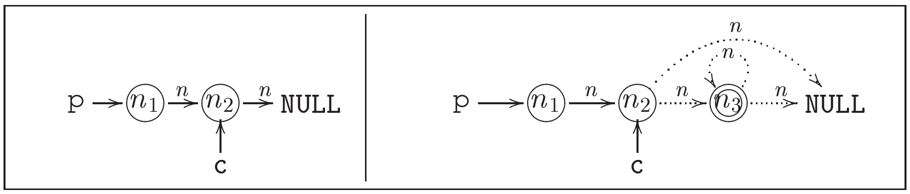

Figure 6.8: The shape graphs that arise after statement c=c→n when the *SG* algorithm is applied to the function shown in Figure 6.1(c) on a shape graph representing a linked list of length two or more elements (g_3 shown in Figure 6.2 without reachability).

programs that are only detected when checking for memory leaks. For example, the program `fumble.c` that erroneously loses part of the list.

- The *PT* algorithm is not precise enough to avoid false alarms of type *ref*. In four out of the ten programs it reports false alarms, most of them on NULL dereference. The *PT* algorithm does not track information about relations between the variables in the program needed in order to correctly analyze conditions and to avoid false alarms.

Cost

Table 6.2 lists the time and space requirements of the implementations on the analyzed programs. The *PT* algorithm is the fastest, running a few milliseconds on all programs. The *SG+R* and *SG* algorithms are similar in the time requirement with one exception, the program `merge.c`. This is the most complicated program to analyze and the *SG+R* algorithm runs significantly faster than the *SG* algorithm.

The time measurement is not a precise comparison since it is dependent on both the PAG and our not very efficient implementation. Although the space and time measurements match one another very well, we regard the space measurement as a more precise comparison. We count the space as the number of total edges (variables and selectors) arising in all points in the program. For the *PT* algorithm an edge corresponds to a points-to pair. In all programs the space requirement was the smallest in the *PT* algorithm.

The space comparison between the *SG* and the *SG+R* is surprising. The worst case space-complexity of the *SG* algorithm is smaller than that of the *SG+R* algorithm. However, the space requirement of the *SG* algorithm is larger than that of the *SG+R* algorithm. In all programs but one the number of edges in the *SG* algorithm is greater than that of the *SG+R* algorithm.

The reason for this extra edges is that in the *SG* algorithm there is less information, it does not contain any explicit information on the reachability from stack variables. This lack of information causes the analysis to be over-conservative and generate shape graphs that are not generated by the

program	*SG+R*		*SG*	
	total	max	total	max
search.c	202	14	217	17
null_deref.c	226	20	241	23
delete.c	296	21	374	26
del_all.c	118	8	144	6
insert.c	301	17	362	19
merge.c	444	46	11416	1393
reverse.c	194	14	202	14
fumble.c	191	14	186	12
rotate.c	148	5	157	6
swap.c	134	5	144	6

Table 6.4: A comparison between the *SG+R* and the *SG* algorithms of the total number of shape graphs (column 1) and the maximal number of shape graphs that arises at a single point in the program (column 2).

SG+R algorithm. For example, consider the program `swap.c` shown in Figure 6.1 and the result of the *SG+R* analysis on this program shown in Figure 6.4. The *SG* algorithm yields the shape graphs shown in Figure 6.8 after statement s_3. This is the first point of the program where the *SG* algorithm yields over-conservative shape graphs that are not generated by the *SG+R* algorithm. The second shape graph has an over-conservative edge from node n_2 to the `NULL` node. This edge will cause the analysis to generate more redundant shape graphs as it progresses through the program.

Additional information used to compare the *SG* and the *SG+R* algorithms is the total number of shape graphs that arise during the analysis and the maximal number of shape graphs generated at a single point in the program. Table 6.4 lists this information[3]. Again, in all programs but one the *SG* generates more shape graphs than the *SG+R* algorithm.

6.3 Conclusion

The usability of cleanness checking tools depends on many factors including: (i) The ability to detect nontrivial bugs; (ii) The simplicity of use; and (iii) The ability to scale for large realistic programs. An easy to use tool should: (i) Report minimal false alarms; (ii) Demand minimal interaction with the user in general, in particular annotations; and (iii) Provide useful error messages.

In this chapter, we focused on the aspect of detection of nontrivial bugs and on ease-of-use. We are encouraged by the fact that *SG+R* can detect bugs in C programs manipulating singly linked lists without any false alarms. Our tool is automatic, it analyzes the program without any interaction with the user and it runs reasonably fast. We use PAG and the XVCG tool [60] for pictorial representation of the shape graphs and the CFG. Lastly, the algorithm has already been generalized to handle many data structures including trees and doubly linked lists [59].

The worst case complexity of our algorithm is quite high. It is linear in the size of the shape graphs. The number of shape nodes is exponential in the number of pointer stack variables and the number of shape graphs is exponential in the number of shape nodes. Yet the average time and space requirements

[3]This information is unavailable for the *PT* algorithm since in this algorithm there is only one set of points-to relations for each point in the program.

of our algorithm depend on the number of different aliasing configurations between program variables. Our initial experience indicates that this number does not necessarily correspond to the program size. For example, it takes 360 CPU seconds to analyze a C program of size 1106 lines, and we generated 20 cleanness messages.

Scaling our techniques for real large programs without generating too many false alarms is a task that will take a significant effort. Examples of techniques that may be useful for this purpose are:

- Procedures' contracts can help, e.g., by requiring that the programmer specifies pre- and post-conditions.

- Better interprocedural analysis can improve the precision and the cost of the analysis. We currently use the automatic call-string approach of PAG to handle procedures. This approach is inadequate since it is both imprecise and expensive.

- Programming languages such as Java simplify cleanness checking by restricting pointer manipulations and by supporting object oriented programming. This can help in improving the efficiency and the effectiveness of static cleanness checking (as opposed to the Java virtual machine which enforces some cleanness at run-time).

- Finally in some cases, practical tools can be developed by ignoring several kinds of errors at the cost of not being conservative. For example, the Prefix tool assumes that procedure parameters are not aliased. This can lead to undetection of certain errors.

Chapter 7

Conclusion

This thesis shows that program analysis can be used to verify the absence of runtime errors with very few false alarms. The absence of such errors can be used to guarantee the quality of the verified programs. Specifically, programs can be ensured against buffer overrun attack which is the most frequently used software attack. The verified C programs are taken from existing (including safety critical) applications and contain complicated aspects of C such as multilevel pointers, destructive updates, and pointer arithmetic. The algorithms have been fully implemented. We also found real errors in public domain legacy heavily used software such as `web2c`.

One of the long standing obstacles in program verification is the difficulty of specifying software behavior. It is very difficult if not impossible to completely specify the behavior of a program. This thesis sidesteps the problem of specifying software behavior by verifying cleanness properties which should exist in any "reasonable" program, independent of the program specification. Most of these cleanness properties are derived from the programming language semantics. Interestingly, our techniques identify cleanness errors which stem from logical errors. An example of such a phenomena in identified by CLLV (see Chapter 6) that reports memory leakage which stems from a logical error in which some list elements are lost.

The CSSV tool described in this thesis advances the state of formal verification in general and that of abstract interpretation in particular— prior to this thesis it was not clear if such a task is at all feasible since existing algorithms were either unsound, produced plethora of false alarms, or did not handle real life programs. This is achieved by combining existing techniques in a novel way and inventing new algorithms. Procedure contracts and advanced program analyses are combined in a novel way to make the whole process feasible on arbitrary C programs.

Specifically, we show how modular analysis can be used to scale the exponential Polyherdra domain [12] which tracks linear relationships between variables in programs with loops in a rather precise way. Also, we show how pointer analysis can be carried in a scalable yet precise way without requiring programmer provided information on pointer aliases.

7.1 Suggestions for Further Work

CSSV requires a sound flow-insensitive partitioning points-to information. This limits the precision of the points-to information and therefore the precision of the entire analysis. Although none of the false messages encountered in our experiments stems from this imprecision, the CSSV algorithm is ineffective for programs manipulating strings in more complicated structures, such as linked lists of strings. For this purpose, a non-partitioning flow-sensitive pointer analysis is required which significantly complicates the problem. In a follow up work described in [30], a method for tracking integer properties of (potentially unbounded) sets of locations is presented.

Contracts exert additional burden on the programmer. Preliminary algorithms for automatically generating pre- and post-conditions is described in [21]. The main idea is to apply side-effect analysis to compute the `modifies` clause. A forward sound integer-analysis algorithm is applied to the integer guarded program to compute an approximation of the strongest postcondition. Similarly, a backward sound integer-analysis algorithm is used to compute an approximation of the weakest liberal precondition [17]. The generated postcondition (precondition) is not necessarily the strongest (weakest) because information is lost during the static integer analysis.

Another solution to the requirement of contracts, is an interprocedural analysis that eliminates the need for contracts but requires scaling up the integer analysis. However, this is not a trivial task. One method of reducing the scaling problem of the integer analysis is to use adaptive domains [4]. Generally, the idea is to use less expensive and less precise domains such as interval analysis and when messages occur apply a more expensive domain such as the polyhedra.

A limitation of static analysis cleanness checking techniques is the reporting of false messages. It can be a difficult task for a programmer to determine whether a reported message reflects a real error. According to our limited experience, it can be very effective in locating the errors since it provides a description of the linear relationships which holds at the point where the error is reported. In [24, 23] a theorem prover is used in order to automatically generate a concrete program input which leads to a violation. Such a tool can be combined with CSSV in order to provide concrete counter examples and to determine whether the error is a false alarm.

Bibliography

[1] L. O. Andersen. Binding-time analysis and the taming of C pointers. In David Schmidt, editor, *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, pages 47–58, New York, NY, 1993. ACM Press.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).

[3] T.M. Austin, S.E. Breach, and G.S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 290–301. ACM Press, 1994.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 196–207, 2003.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 321–333, 2000.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[7] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[8] B. Chess. Improving computer security using extended static checking. In *IEEE Symposium on Security and Privacy*, pages 160–176, 2002.

[9] Compaq Systems Research Center. Compaq extended static checker for Java. Available at " http://research.compaq.com/SRC/esc/", 2000.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Symp. on Princ. of Prog. Lang.*, pages 84–96, 1978.

[13] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *In Proc. of the DARPA Information Survivability Conference and Expo*, 1999.

[14] M. Das. Unification-based pointer analysis with directional assignments. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 35–46, 2000.

[15] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.

[16] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *SIGPLAN conf. on Lang. Comp. and Tool for Embedded Sys.*, 2003.

[17] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[18] N. Dor, M. Rodeh, and M. Sagiv. Detecting memory errors via static pointer analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 27–34, June 1998.

[19] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symp.*, pages 115–134. Springer, 2000.

[20] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symp.*, pages 194–212, 2001.

[21] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 155–167, 2003.

[22] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, New York, NY, 1994. ACM Press.

[23] G. Erez. Generating concrete counter examples for arbitrary abstract domains. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2004. In Preparation.

[24] G. Erez, M. Sagiv, and E. Yahav. Generating concrete counter examples for arbitrary abstract domains. Unpublished Manuscript, 2003.

[25] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 44–53, 1996.

[26] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Symp. on Princ. of Prog. Lang.*, pages 379–392, January 1995.

[27] P. Fradet, R. Gaugne, and D. Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *Proc. European Symposium on Programming, ESOP'96 , LNCS*, pages 125–240, 1996.

[28] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Symp. on Princ. of Prog. Lang.*, pages 121–133, New York, NY, 1998. ACM Press.

[29] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 47–58, 2001.

[30] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Conf. on Tools and Algo. for the Construction and Analysis of Syst.*

[31] Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702, pages 274–309. Springer, 1993.

[32] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[33] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 254–263, 2001.

[34] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411. ACM, New York, May 1992.

[35] IBM. Rational purifyplus. Available at "http://www-306.ibm.com/software/awdtools/purifyplus", 2003.

[36] D. Jackson. Aspect, an economical bug detector. In *Proceedings of the 13th International Conference on Software Engineering*, pages 13–22, May 1994.

[37] B. Jeannet. New polka library. Available at "http://www-verimag.imag.fr/∼bjeannet/newpolka-english.html".

[38] J.L. Jensen, M.E. Joergensen, N.Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 226–236, 1997.

[39] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[40] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1988.

[41] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices*, 30(6):270–278, 1995.

[42] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, pages 45–51, 2001.

[43] G. Leavens and A. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *Formal Methods*, pages 1087–1106, 1999.

[44] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symp.*, pages 279–298, 2001.

[45] A. Loginov, S. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Proc. of Fundamental Approaches to Softw. Eng. (FASE)*, pages 217–232, April 2001.

[46] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[47] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of Unix utilities and services, 1995. Available at http://www.cs.wisc.edu/∼bart/fuzz/fuzz.html.

[48] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[49] E.W. Myers. A precise inter-procedural data flow algorithm. In *Symp. on Princ. of Prog. Lang.*, pages 219–230, 1981.

[50] G. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Symp. on Princ. of Prog. Lang.*, 2002.

[51] F. Nielson and H. R. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992.

[52] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[53] M. Norrish. *Formalising C in HOL*. PhD thesis, Computer Lab., University of Cambridge, 1998.

[54] Cop. Parasoft. Insure++. Available at “http://www.parasoft.com/”, 1999.

[55] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *Proc. European Symp. on Programming*, pages 380–398, 2003.

[56] R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 182–195, 2000.

[57] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. Tech. Rep. TR-1383, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, July 1998. Available at “http://www.cs.wisc.edu/wpis/papers/parametric.ps”.

[58] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.

[59] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, pages 105–118, 1999.

[60] G. Sander. Graph layout through the vcg tool. *Graph Drawing, DIMACS International Workshop GD’94*, pages 194–205, 1995.

[61] B. Scholz, J. Blieberger, and T. Fahringer. Symbolic pointer analysis for detecting memory leaks. In *SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation*, pages 103–113, 2000.

[62] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symp. on Princ. of Prog. Lang.*, pages 1–14, 1997.

[63] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[64] A. Simon and A. King. Analyzing string buffers in c. In *International Conference on Algebraic Methodology and Software Technology*, pages 365–379, 2000.

[65] E. Spafford. The internet worm: Crisis and aftermath. In *Communications of the ACM, June 1989. 165*, pages 678–687, June 1989.

[66] J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Inf. and Comp.*, 101(1):70–102, Nov. 1992.

[67] SUIF. Suif compiler system. Available at "http://suif.stanford.edu/".

[68] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symp. on Network and Distributed Systems Security*, 2000.

[69] G. Yorsh. *CoreC: A Generic Frontend for C*, 2002. http://www.cs.tau.ac.il/∼ gretay.

[70] B. Zorn and P. Hilfinger. A memory allocation profilers for c and lisp programs. In *In Proceedings of Summer USENIX'88 Conference*, pages 223–237, 1988.

Appendix A

Proofs

A.1 Proof of Expression Preserving of Sound APT

Lemma 4.2.6 *If an abstract points-to state $\sigma^{APT} = (\mathcal{BA}, loc, pt, count, allocatedAt)$, is a sound approximation of program fragment P then:*

for each admissible concrete state $\sigma^{\natural}$ arising during the execution of P and for each expression $E \in$ **Expr***:*

$$\alpha(lv(E, \sigma^{\natural})) \in lv_{pt}(E)$$

Proof. Since σ^{APT} is a sound approximation of a program fragment then for every state $\sigma^{\natural}$ there exists an α mapping for which σ^{APT} is a sound approximation. We prove the lemma by rule induction on all expressions E, for a given execution and an α mapping.

variable $E ::= v$ We need to show that

$$\alpha(lv(v, \sigma^{\natural})) \in lv_{pt}(v)$$

By the definition of concrete and abstract L-value

$$\alpha(loc^{\natural}(v, depth^{\natural})) \in loc(v)$$

Indeed, according to the first condition for a sound abstract pointer information (in Definition 4.2.3) $\alpha(loc^{\natural}(\sigma^{\natural})(v, i)) \in loc(v)$ for all i.

dereference $E ::= *N$ We have the following definition

$$\begin{array}{l} lv(E, \sigma^{\natural}) = content^{\natural}(\sigma^{\natural})(lv(N, \sigma^{\natural})) \\ lv_{pt}(E) = pt(lv_{pt}(N)) \end{array}$$

Form the recursion assumption we conclude that

$$\alpha(lv(N, \sigma^{\natural})) \in lv_{pt}(N)$$

Define $l_1^{\natural}, l_2^{\natural}, l_1, l_2$ as follows:

$$\begin{array}{lcl} l_1^{\natural} & \stackrel{\text{def}}{=} & lv(N, \sigma^{\natural}) \\ l_2^{\natural} & \stackrel{\text{def}}{=} & lv(*N, \sigma^{\natural}) = content^{\natural}(\sigma^{\natural})(lv(N, \sigma^{\natural})) \\ l_1 & \stackrel{\text{def}}{=} & \alpha(l_1^{\natural}) \\ l_2 & \stackrel{\text{def}}{=} & \alpha(l_2^{\natural}) \end{array}$$

By substitution, we have

$$l_1 = \alpha(l_1^\natural) \in lv_{pt}(N)$$

We need to show that,

$$\begin{aligned} & \alpha(lv(*N, \sigma^\natural)) \in lv_{pt}(*N) \\ \equiv \quad & \alpha(l_2^\natural) \in pt(lv_{pt}(N)) \end{aligned}$$

Since $l_1 \in lv_{pt}(N)$, it is enough to show that

$$l_2 = \alpha(l_2^\natural) \in pt(l_1)$$

Indeed, according to the second requirement of a sound APT, we conclude that

$$l_2 = \alpha(l_2^\natural) \in pt(\alpha(l_1^\natural)) = pt(l_1)$$

field operator $E ::= E'.f$ We have the following definition

$$\begin{aligned} & lv(E, \sigma^\natural) = lv(E') + \mathit{field\_pos}(f) \\ & lv_{pt}(E) = lv_{pt}(E') \end{aligned}$$

From the definition of base address, we conclude that

$$base^\natural(\sigma^\natural)(lv(E', \sigma^\natural)) = base^\natural(\sigma^\natural)(lv(E', \sigma^\natural) + \mathit{field\_pos}(f))$$

From the requirement of same abstraction of locations with the same base address

$$\alpha(lv(E', \sigma^\natural)) = \alpha(lv(E', \sigma^\natural) + \mathit{field\_pos}(f))$$

Therefore,

$$\begin{aligned} & \alpha(lv(E', \sigma))) \in lv_{pt}(E') && \text{Recusion assumption} \\ \equiv \quad & \alpha(lv(E', \sigma) + \mathit{field\_pos}(f)) \in lv_{pt}(E') && \text{Definitoin of base address} \\ \equiv \quad & \alpha(lv(E, \sigma)) \in lv_{pt}(E) && \text{Substitution} \end{aligned}$$

■

A.2 Proof of Soundness of *PPT* Extraction

Lemma 4.2.7 *For every formal parameter f and an admissible state at P, $\sigma^\natural$, and a sound approximation of $\sigma^\natural$,*

$$\sigma^P = (\mathcal{BA}_P, loc_P, pt_P, count_P, allocatedAt_P)$$

with respect to a mapping function $\alpha \colon \mathcal{BA}^\natural(\sigma^\natural) \to \mathcal{BA}_P$, if `Parameterizable`$(\sigma^P, f)$ returns true then at most one of the reachable locations in $\mathcal{L}^\natural(\sigma^\natural)$ is represented by the abstract locations pointed to by f.

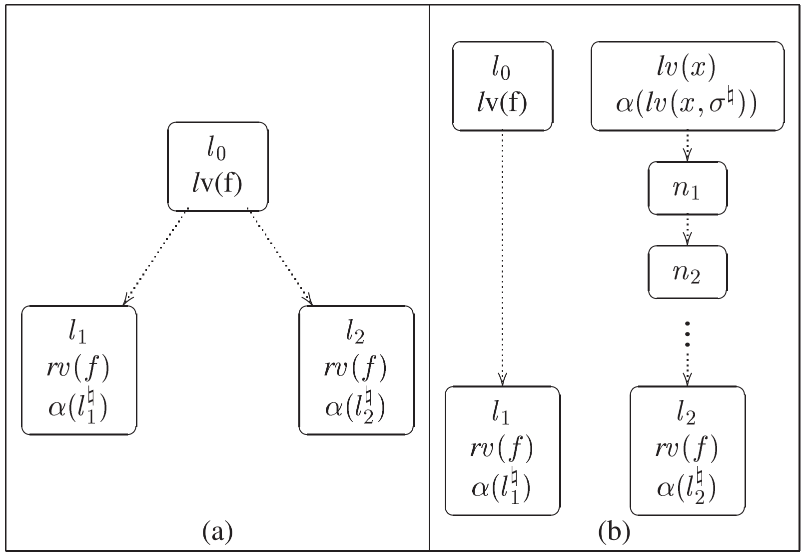

Figure A.1: A points-to information at entry to `Parameterize` (a), and pt' on the first iteration (b) .

Proof. We prove the lemma by showing that for two reachable locations represented by abstract locations pointed-to by a formal, the function `Parameterizable` must return false.

Assume a state $\sigma^\natural$ in which $l_1^\natural, l_2^\natural \in \mathcal{L}^\natural(\sigma^\natural)$ are different reachable locations,
represented by abstract locations pointed-to by f:

$$\begin{array}{l} l_1^\natural \neq l_2^\natural \\ \alpha(l_1^\natural), \alpha(l_2^\natural) \in rv_{pt}(f, \sigma^P) \end{array}$$

Define l_0, l_1, l_2 as follows

$$\begin{array}{ll} l_0 & \stackrel{\text{def}}{=} \alpha(loc^\natural(f)) \\ l_1 & \stackrel{\text{def}}{=} \alpha(l_1^\natural) \\ l_2 & \stackrel{\text{def}}{=} \alpha(l_2^\natural) \end{array}$$

Figure A.1(a) graphically displays this configuration.

Since the two locations, $l_1^\natural, l_2^\natural$, are reachable and each expression's R-value is a single value, there must be two different expressions e_1, e_2 such that their R-value are the given locations:

$$\begin{array}{l} e_1 \neq e_2 \\ rv(e_1, \sigma^\natural) = l_1^\natural \\ rv(e_2, \sigma^\natural) = l_2^\natural \end{array}$$

We need to show that `Parameterizable`(σ^P, f) returns false. Since e_1 and e_2 are two different expressions, at least one of them must be different then the formal variable f. Assume, without loss of the generality, that $e_2 \neq f$. Assume that e_2 starts with a visible variable, x, e.g., $e_2 = x + 6$ or $e_2 = *x$. Note that x may be f.

On the first iteration of the loop in `Parameterizable`, the edge from l_0 to l_2 is removed. We need to show that l_2 is still reachable from the PT-L-value of f in the new points-to graph. Let us mark the edges on the path from $\alpha(lv(x, \sigma^\natural))$ to l_2 as $E = \{e_0, e_2, \cdots, e_m\}$, where:

$$\begin{array}{lcl} e_0 & \stackrel{\text{def}}{=} & (\alpha(lv(x,\sigma^\natural)), n_1) \\ e_i & \stackrel{\text{def}}{=} & (n_i, n_{i+1}) \text{ for } 0 < i < m \\ e_m & \stackrel{\text{def}}{=} & (n_{m-1}, l_2) \end{array}$$

Figure A.1(b) displays this path in the existing configuration.

We claim that the edge from (l_0, l_2) is not in E and therefore l_2 is still reachable from x.

Assume that the contrary holds, i.e., $(l_0, l_2) \in E$ thus there exists an i such that $(l_0, l_2) = e_i$. There are two possible cases for the value of i.

- $i = 1$: This implies that the path is $E = \{(l_0, l_2)\}$ yielding that $E_2 = f$ which contradicts our assumption.
- $0 < i \leq m$: Since there is an edge from l_0 to l_2 there exists an edge from l_{i-1} to l_0. This edge can appear only if the address-of formal parameter f has been taken which is forbidden by CoreC. Again, we have reached a contradiction.

Therefore, edge (l_0, l_2) is not in E, indicating that l_2 is still reachable from x. The algorithm `Param-eterizable` returns false. ■

A.3 Proof of Soundness of CSSV

Lemma 4.4.1 *Given a pair of concrete and integer state $(\sigma^\natural, \sigma^{\mathcal{I}}) \in \varphi$, i.e., $\sigma^{\mathcal{I}}$ is a simulation of $\sigma^\natural$, then for every expression $E \in$ **Expr** such that $rv(E, \sigma) \notin \{uninit, undef\}$ the following holds:*

$$\begin{array}{ll} rv(E, \sigma^\natural) \in eval(Tr(E), \sigma^{\mathcal{I}}) & \textit{where type(E) = integer} \vee \\ rv(E.offset, \sigma^\natural) \in eval(Tr(E), \sigma^{\mathcal{I}}) & \textit{where type(E) = t*} \end{array}$$

Proof: By rule induction on E.

- Constant: $E ::= c$

  This case is rather straightforward.

  $$\begin{array}{lcll} rv(c, \sigma^\natural) & = & c & \text{From the definition of R-value in Table 2.4} \\ eval(Tr(c), \sigma^{\mathcal{I}}) & = & eval(c, \sigma^{\mathcal{I}}) = \{c\} & \text{Transformation rules} \end{array}$$

  Clearly, $c \in \{c\}$.

- simple expression ($E ::= N$) or a dereference expression ($E ::= *E'$)

  For these cases the definition of R-value is:

  $$rv(E, \sigma^\natural) = content^\natural(lv(E, \sigma^\natural))$$

  Define $l^\natural$ to be $lv(E, \sigma^\natural)$.

  From Lemma 4.2.6,

  $$\alpha(lv(E, \sigma^\natural)) \in lv_{pt}(E)$$

  From the definition of location simulation, simulating formula $\psi(l^\natural, b)$ holds when $b = \alpha(l^\natural)$.

  We have here two cases:

- E is an integer expression.
  From the simulation formula if $\mathit{content}^\natural(\sigma^\natural)(l^\natural) \in \mathit{primitive}$ then

$$\mathit{content}^\natural(\sigma^\natural)(l^\natural) \in eval(b.\mathit{val}, \sigma^{\mathcal{I}})$$

Therefore, it is enough to show that:

$$eval(b.\mathit{val}, \sigma^{\mathcal{I}}) \subseteq eval(\mathit{Tr}(E), \sigma^{\mathcal{I}})$$

Indeed,

$$\begin{array}{lll} \alpha(\mathit{lv}(E, \sigma^\natural)) & \in \mathit{lv}_{pt}(E) & \text{Sound PPT} \\ b & \in \mathit{lv}_{pt}(E) & \text{since } b = \alpha(l^\natural) \end{array}$$

Therefore, we can conclude that

$$\begin{array}{lll} eval(b.\mathit{val}, \sigma^{\mathcal{I}}) & \subseteq eval(\mathit{lv}_{pt}(E).\mathit{val}, \sigma^{\mathcal{I}}) & \\ & = eval(\mathit{Tr}(E), \sigma^{\mathcal{I}}) & \text{transformation rules} \end{array}$$

- E is a pointer expression.
  We need to how that

$$\mathit{rv}(E.\mathit{offset}, \sigma^\natural) = \mathit{offset}^\natural(\sigma^\natural)(\mathit{content}^\natural(\sigma^\natural)(\mathit{lv}(E))) \in eval(\mathit{Tr}(E), \sigma^{\mathcal{I}})$$

We have defined $l^\natural$ to be lv(E). From the simulation formula if $\mathit{content}^\natural(\sigma^\natural)(l^\natural) \in \mathcal{L}^\natural(\sigma^\natural)$ then

$$\mathit{offset}^\natural(\sigma^\natural)(\mathit{content}^\natural(\sigma^\natural)(l^\natural)) \in eval(b.\mathit{offset}, \sigma^{\mathcal{I}})$$

Therefore, it is enough to show that:

$$eval(b.\mathit{offset}, \sigma^{\mathcal{I}}) \subseteq eval(\mathit{Tr}(E), \sigma^{\mathcal{I}})$$

Similarly, to the scalar case:

$$\begin{array}{ll} \alpha(\mathit{lv}(E, \sigma^\natural)) \in \mathit{lv}_{pt}(E) & \\ b \in \mathit{lv}_{pt}(E) & \text{since } b = \alpha(l^\natural) \end{array}$$

Therefore, we can conclude that

$$\begin{array}{lll} eval(b.\mathit{offset}, \sigma^{\mathcal{I}}) & \subseteq eval(\mathit{lv}_{pt}(E).\mathit{offset}, \sigma^{\mathcal{I}}) & \\ & = eval(\mathit{Tr}(E), \sigma^{\mathcal{I}}) & \text{Transformation rules} \end{array}$$

- Cast $E ::= (t)E'$

  We need to distinguish between the different type of casting. Casting from one integer-based type to another integer-base type results in the same value. For this case we have the following:

$$\begin{array}{ll} \mathit{rv}(E, \sigma^\natural) = \mathit{rv}(E', \sigma^\natural) & \text{From the definition of R-value} \\ \mathit{rv}(E', \sigma^\natural) \in eval(\mathit{Tr}(E'), \sigma^{\mathcal{I}}) & \text{From the induction assumption} \\ eval(\mathit{Tr}(E), \sigma^{\mathcal{I}}) = eval(\mathit{Tr}(E'), \sigma^{\mathcal{I}}) & \text{From the transformaiton rules} \end{array}$$

Therefore we can conclude that

$$\mathit{rv}(E, \sigma^\natural) \in eval(\mathit{Tr}(E), \sigma^{\mathcal{I}})$$

A similar case is casting from one type of pointer to another. In this case $type(E')$ is a pointer type. Therefore from the induction assumption:

$$rv(E'.offset, \sigma^\natural) \in eval(Tr(E'), \sigma^\mathcal{I})$$

The offset property is type independent. Thus

$$rv(((t)E').offset, \sigma^\natural) = rv(E'.offset, \sigma^\natural)$$

In addition, the transformation rules implies that $Tr(E) = Tr(E')$ Therefore, we can conclude that

$$rv(E, \sigma^\natural) = rv(E'.offset, \sigma^\natural) \in eval(Tr(E'), \sigma^\mathcal{I}) = eval(Tr(E), \sigma^\mathcal{I})$$

Other casting operations result in $\top$. Therefore

$$rv(E, \sigma^\natural) \in \top = eval(Tr(E), \sigma^\mathcal{I})$$

- Address-of $E ::= \&E'$

  The address of operator is always a pointer expression. Therefore, we need to show that

  $$rv(E.offset, \sigma^\natural) \in eval(Tr(E), \sigma^\mathcal{I})$$

  We show this by rule induction on E'

  variable $E' :== v$ This is the simple case:

  $$rv(\&v.offset, \sigma^\natural) = 0 \in eval(0, \sigma^\mathcal{I}) = \{0\}$$

  dereference $E' :== *N$ This is based on the fact that $\&(*N)$ is simplified to N.

  $$\begin{array}{lll} rv(N.offset, \sigma^\natural) & \in eval(Tr(N), \sigma^\mathcal{I}) & \text{Induction assumption} \\ rv(\&(*N).offset, \sigma^\natural) & \in eval(Tr(\&(*N)), \sigma^\mathcal{I}) & \text{Basic operator rules} \\ rv(E.offset, \sigma^\natural) & \in eval(Tr(E), \sigma^\mathcal{I}) & \end{array}$$

  field $E' :== E''$**.f** We observe the following

  $$\begin{array}{ll} rv(\&E''.offset, \sigma^\natural) \in eval(Tr(\&E''), \sigma^\mathcal{I}) & \text{Induction assumption} \\ rv(\&(E'').offset + field\_pos\mathrm{f}, \sigma^\natural) \in eval(Tr(\&E'') + field\_pos\mathrm{f}, \sigma^\mathcal{I}) & \text{Basic arithmetic rules} \\ rv(\&(E'').offset + field\_pos\mathrm{f}, \sigma^\natural) \in eval(Tr(\&(E''.\mathrm{f})), \sigma^\mathcal{I}) & \text{From transforamtion rules} \\ rv(\&(E''.\mathrm{f}).offset, \sigma^\natural) \in eval(Tr(\&(E''.\mathrm{f})), \sigma^\mathcal{I}) & \text{From definition of R-value} \\ rv(E.offset, \sigma^\natural) \in eval(Tr(E), \sigma^\mathcal{I}) & \end{array}$$

- Binary operator $E ::= N_1 \; op \; N_2$

  Let us first consider the case that N_1 and N_2 are both integer expressions. From the induction assumption

  $$rv(E_1, \sigma^\natural) \in eval(Tr(E_1), \sigma^\mathcal{I})$$
  $$rv(E_1, \sigma^\natural) \in eval(Tr(E_2), \sigma^\mathcal{I})$$

  Therefore

  $$\begin{array}{rcll} rv(E_1 \; op \; E_2, \sigma^\natural) & = & rv(E_1, \sigma^\natural) \; op \; rv(E_2, \sigma^\natural) & \text{Definition of R-value} \\ & \in & eval(Tr(E_1), \sigma^I) \; op \; eval(Tr(E_2), \sigma^I) & \text{Rule induction} \\ & = & eval(Tr(E_1 \; op \; b_2), \sigma^I) & \text{Transfomation rules} \end{array}$$

First, consider the case of two pointer expression, thus E_1 and E_2 are pointer expression. However, E is an integer expression. We need to observe the following semantic equivalence

$$\begin{array}{lll} E_1 \ op \ E_2 & \equiv (E_1.\mathit{offset} + E_1.base) \ op \ (E_2.\mathit{offset} + E_2.base) & \text{Definition of pointer values} \\ & \equiv E_1.\mathit{offset} \ \ op \ E_2.\mathit{offset} & \text{Since } E_1.base = E_2.base \\ & & \text{(preconidtion for pointer operations)} \end{array}$$

Therefore we need to show that

$$rv(E_1.\mathit{offset} \ \ op \ E_2.\mathit{offset}, \sigma^\natural) \in eval(Tr(E_1 \ op \ E_2), \sigma^I)$$

Indeed,

$$\begin{array}{llll} rv(E_1.\mathit{offset} \ \ op \ E_2.\mathit{offset}, \sigma^\natural) & = & rv(E_1.\mathit{offset}, \sigma^\natural) \ op \ rv(E_2.\mathit{offset}, \sigma^\natural) & \text{Definition of R-value} \\ & \in & eval(Tr(E_1), \sigma^I) \ op \ eval(Tr(E_2), \sigma^I) & \text{Rule induction} \\ & = & eval(Tr(E_1 \ op \ b_2), \sigma^I) & \text{Transfomation rules} \end{array}$$

Finally, we need to consider the case of $\mathit{type}(E_1) = t*$ is a pointer type and E_2 is an integer type.

$$\begin{array}{lll} rv(E.\mathit{offset}, \sigma^\natural) & = rv(E_1.\mathit{offset}, \sigma^\natural) \ op \ rv(E_2, \sigma^\natural) rv(N_2) * \mathit{sizeof}(\mathit{type}(*N_1)) & \\ \in & eval(Tr(E_1), \sigma^{\mathcal{I}}) \ op \ eval(Tr(E_2), \sigma^{\mathcal{I}}) * \mathit{sizeof}(\mathit{type}(*N_1)) & \text{Induction assumption} \\ = & eval(Tr(E_1) \ op \ Tr(E_2) * \mathit{sizeof}(\mathit{type}(*N_1)), \sigma^{\mathcal{I}}) & \text{Basic operation} \\ = & eval(Tr(E_1 \ op \ E_2), \sigma^{\mathcal{I}}) & \text{Transformation rules} \\ = & eval(Tr(E), \sigma^{\mathcal{I}}) & \end{array}$$

- Base attribute $E :== E'$.base

  This is a pointer expression. Therefore we need to show that

  $$rv(E.\mathit{offset}, \sigma^\natural) \in eval(Tr(E), \sigma^{\mathcal{I}})$$

  Indeed,

  $$\begin{array}{llll} rv((E'.base).\mathit{offset}, \sigma^\natural) & = & 0 & \text{Offset of any base address is zero} \\ eval(Tr(E'.base), \sigma^{\mathcal{I}}) & = & eval(\{0\}, \sigma^{\mathcal{I}}) & \text{Transformation rules} \\ & = & \{0\} & \end{array}$$

- Offset attribute: $E :== E'$.offset This is an integer expression. Therefore we need to show that

  $$rv(E, \sigma^\natural) \in eval(Tr(E), \sigma^{\mathcal{I}})$$

  let us mark $l^\natural = lv(E', \sigma^\natural)$ and $b = \alpha(l^\natural)$. The location-simulation formula requires that

  $$\mathit{offset}^\natural(\sigma^\natural)(\mathit{content}^\natural(\sigma^\natural)(l^\natural)) \in eval(b.\mathit{offset}, \sigma^{\mathcal{I}})$$

  From this we can conclude that

  $$\begin{array}{lll} rv(E'.\mathit{offset}, \sigma^\natural) & = \mathit{offset}^\natural(\sigma^\natural)(rv(E')) & \text{R-value of attributes} \\ & = \mathit{offset}^\natural(\sigma^\natural)(\mathit{content}^\natural(\sigma^\natural)(l^\natural)) & \text{Since } l^\natural = lv(E', \sigma^\natural) \\ & \in eval(b.\mathit{offset}, \sigma^{\mathcal{I}}) & \text{Location-simulation requirement} \\ & \subseteq eval(lv_{pt}(E').OFFSET, \sigma^{\mathcal{I}}) & \text{Since } b \in lv_{pt}(E) \\ & = eval(Tr(E'.\mathit{offset}), \sigma^{\mathcal{I}}) & \text{Transformaiton rules} \end{array}$$

- Alloc attribute: $E :== E'$.alloc We need to show that

  $$rv(E'.\texttt{alloc}, \sigma^\natural) \in eval(Tr(E'.\texttt{alloc}), \sigma^\mathcal{I})$$

  let us mark $l^\natural = rv(E', \sigma^\natural)$ and $b = \alpha(l^\natural)$. The location-simulation formula requires that

  $$size^\natural(\sigma^\natural)(base^\natural(\sigma^\natural)(l^\natural))) \in eval(b.size, \sigma^\mathcal{I})$$

  From the semantics definition:

  $$rv(E'.alloc, \sigma^\natural) = size^\natural(\sigma^\natural)(base^\natural(\sigma^\natural)(l^\natural)) - offset^\natural(\sigma^\natural)(l^\natural)$$

  The transformation rules is:

  $$Tr(E'.alloc) = rv_{pt}(E').SIZE - lv_{pt}(E').OFFSET$$

  Therefore we can conclude that

  $$\begin{array}{lll} rv(E'.alloc, \sigma^\natural) & = size^\natural(\sigma^\natural)(base^\natural(\sigma^\natural)(l^\natural)) - offset^\natural(\sigma^\natural)(l^\natural) & \\ & \in eval(b.size, \sigma^\mathcal{I}) - offset^\natural(\sigma^\natural)(l^\natural) & \text{Location-simulaiton} \\ & \subseteq eval(b.size, \sigma^\mathcal{I}) - eval(lv_{pt}(E').OFFSET, \sigma^\mathcal{I}) & \text{Induction assumption} \\ & \subseteq eval(lv_{pt}(E').SIZE - lv_{pt}(E').OFFSET, \sigma^\mathcal{I}) & \text{Soundness of } PPT \\ & = eval(Tr(E'.alloc), \sigma^\mathcal{I}) & \end{array}$$

- Other attributes

  The other attributes are proved in a similar manner and are strongly based on the induction assumption.