Fully Automatic Verification of Absence of Errors via Interprocedural Integer Analysis

Ron Ellenbogen

December, 2004

Acknowledgements

I would like to express my sincere gratitude to all those who contributed to the completion of my thesis.

First, I would like to express my deep gratitude, respect, and thanks to Dr. Shmuel (Mooly) Sagiv, my supervisor. This thesis would have been impossible to complete without his constant encouragement, guidance, patience and support.

I thank Dr. Nurit Dor for her guidance and support. Nurit's extensive knowledge of the issues researched in this thesis, as well as her willingness to help and to share her knowledge, were a great help in the completion of this thesis.

I am also grateful for the help of Prof. Roberto Bagnara in the implementation of the iCSSV tool. Without Roberto's help concerning the correct usage of the Parma Polyhedra library and his great enthusiasm towards developing the implementation, iCSSV would not have been able to run on the programs it runs on today.

To my colleagues from the Tel-Aviv Programming Languages group: Roman Manevich, Greta Yorsh, and Noam Rinetzky, for their ideas, support and cheerfulness. To Izhakian Zur and Nir Andelman for all of the over lunch discussions concerning University, research, politics and life.

And last but not least, to all of my family. Especially I thank my parents who supported me throughout my studies, and my wife Gitit whose encouragement kept me going through the long hours of writing and debugging.

This research was supported by grants from the Israeli Academy of Science.

Abstract

We present a interprocedural C String Static Verifier (iCSSV), a whole program analysis algorithm for verifying the safety of string operations in C programs. The algorithm automatically proves linear relationships among pointer expressions. The algorithm is conservative, i.e., it infers only valid relationships although it may fail to detect some of them. The algorithm is targeted to programs with "shallow" pointers and complex integer relationships. Therefore, the algorithm combines context-sensitive flow-insensitive pointer analysis of pointer updates with context-sensitive and flow-sensitive integer analysis of properties of allocation sites. Context-sensitivity is achieved by specializing pointer aliases to the context and functional integer analysis.

The algorithm is powerful enough to verify the absence of string manipulation errors such as accesses beyond buffer length and null terminating character. Here the interprocedural analysis guarantees that our algorithm is fully automatic, i.e., does not require user annotations or any other intervention.

A prototype of the algorithm was implemented. Several novel techniques are employed to make the interprocedural analysis of realistic programs feasible.

Contents

Chapter 1

Introduction

1.1 Motivation

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 have caused loss of time, money, and which sometimes penetrate the privacy of users. Across all operating systems and all programming languages, it is up to the programmer to write a safe and reliable code [15, 30]. Some languages, such as Java, are more secure where a runtime error can mostly cause a denial-of-service. In other languages, such as C, errors are more dangerous as they can provide a mechanism for attackers to gain privileges over the system.

Many software defects result from a misuse of strings and standard library functions (e.g., `gets(), strcpy()`) which are inherently unsafe. The programmer is responsible for checking that basic operations cannot *overrun* 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 ($'\backslash 0'$). 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. Although the awareness of the string vulnerability has increased and companies have put a significant amount of resource in fixing existing code, new vulnerabilities are found and exploited, see [4] for a typical string concatenation.

Dynamic testing techniques can aid in finding errors, e.g., [1, 16, 24]. However, their effectiveness depends on the chosen inputs and they can never verify the absence of errors. Due to these limitations there is an increasing demand for static techniques that can detect all errors and provide a code quality assurance against specific error types. Indeed, many interesting static analysis algorithms and tools have been developed to aid in finding defects, e.g., [20, 12]. Furthermore, the *conservative* approach takes into account all possible inputs and behaviors, and thus does not miss any violation. By proving the absence of errors, 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. However, this code quality assurance is achieved at a cost of sometimes generating superfluous messages. To avoid as many superfluous messages as possible, the static analysis must infer complex conditions that arise during the program execution. It is a challenging task to define an abstraction of the program's runtime state that is efficient both in terms of scalability as well as in its small percentage of superfluous messages. One of the main challenges in such static analyses is the handling of procedure calls.

1.2 Main Results

In [13], a conservative static analysis algorithm that uncovers all potential string violations in C programs was presented. Thus, the algorithm can prove the absence of string violations in C programs. The algorithm was successfully applied to Avionic software. However, the algorithm presented there is not fully automatic — it requires procedure contracts defining pre- and post-conditions for every procedure. Therefore, the algorithm cannot be applied to legacy software. Moreover, many programmers are reluctant to define contracts.

This thesis presents a fully automatic algorithm for proving the absence of string errors using a new interprocedural analysis. This paper presents a new interprocedural pointer and integer analysis that computes for every procedure a summary information according to calling contexts. A new C string analysis algorithm is defined using this interprocedural analysis. It can verify the absence of string-related errors, including read or write beyond the bound of a buffer, missing null-termination or insecure calls to the standard C library. This algorithm compares favorably with the existing algorithm due to its automation and precision, see Section 6. Since user annotations or procedural contracts are not required, the analysis is more usable for the typical programmer. Due to the rather precise integer analysis, algorithm verification of the absence of errors is achieved, even in complex conditions, such as in the case of string concatenation. A prototype implementation was used to check this new algorithm to detect string manipulation errors. Our results indicate that the interprocedural analysis is feasible in terms of cost-effectiveness, number of false messages and ease of use. We elaborate on the theoretical and technical contributions in the following subsections.

1.2.1 Interprocedural Integer and Pointer Analysis

Our analysis proves linear relationships between integer properties of pointer expressions. The analysis does not require any annotations or specifications of conditions from the user. This automation is achieved by combining global pointer analysis with an inter-procedural pointer and integer analysis. Procedure calls are handled by summarizing the effect of a procedure for

each given *pointer aliasing* configuration.

1.2.2 String Analysis

Our algorithm translates string manipulations to integer operations. Each operation involving a string, such as string references, updates and manipulations through library routines, is translated to operations over integer variables associated with each string, that represent the size of the buffer allocated for the string, whether the buffer contains a null-terminated string or not, and the length of the string. After this translation is performed, safety conditions such as "A pointer to a string never overflows" can be verified.

1.2.3 Empirical Results

We have implemented iCSSV using the AST-Tooklit [23], CoreC, the Golf pointer analysis [9, 10], and the polyhedra integer analysis of [6] from [2]. We have applied the implementation to real-life programs. iCSSV was run on the application fixwrites, part of web2c and on apr_getpass part of apache's apr libraries.

1.2.4 Outline of this Thesis

The rest of this thesis is organized as follows. Chapter 2 provides an overview of the interprocedural analysis by presenting the derived string analysis on an example. Chapter 3 presents the translation from C program to procedural integer program. Chapter 4 presents the interprocedural analysis. Chapter 5 describes the prototype implementation and the empirical results. Related work is discussed in Chapter 6. Chapter 7 concludes this thesis and presents ideas for further work.

Chapter 2

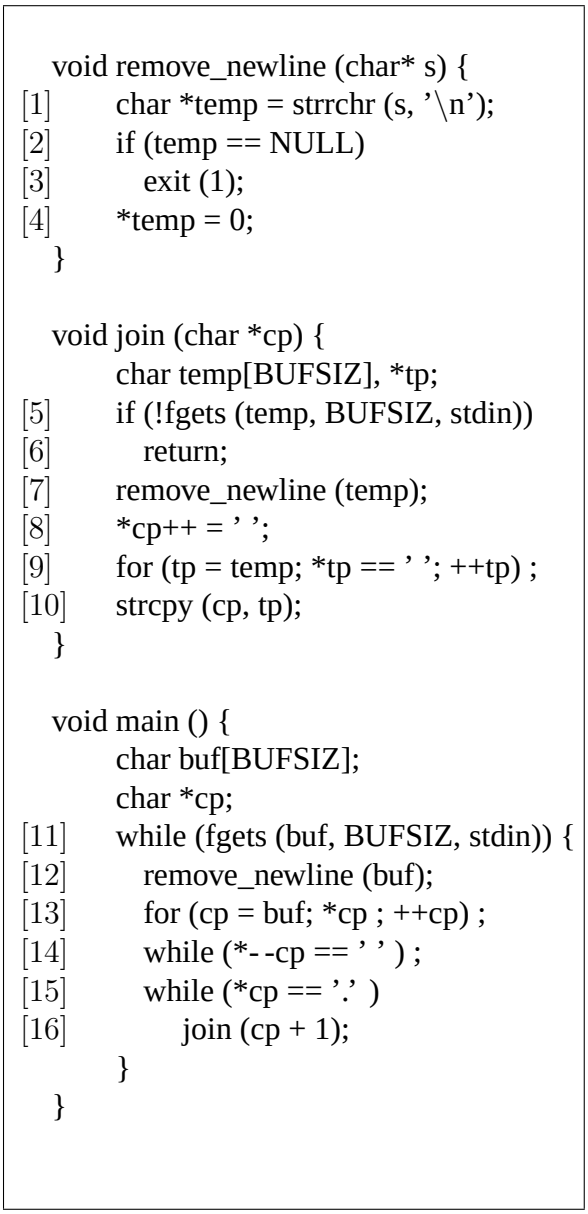
Overview

This chapter presents our interprocedural analysis informally by showing how the derived C string analysis works on an example. Consider the code section, based on `fixwrites` shown in Fig. 2.1. This program reads input, and while the input ends with a '.' another input is concatenated. Note that the procedure `remove_newline` is called with different pointer aliasing configurations. The call on line [12] is with different memory-aliasing on the first and proceeding iterations of the containing while-loop. On the call during the first iteration there is no *overlapping* between `buf` and `cp`. During other calls on line [12] `buf` and `cp` overlap, i.e., they point within the same array but at potentially different offsets. The memory aliasing configuration on the call to `remove_newline` from line [7] is different and involves only the local `temp` array.

Our analysis detects two errors in this program. The first error reports that the `cp` pointer may underflow at line [14]. This results from the fact that the program unsafely assumes the existence of a non-space character in the input stream. The second error reported concerns a potential buffer overrun during the call to `strcpy()` at line [10] in `join`, due to the fact that `cp` may not point to sufficient memory space to copy the string in `temp`. Other string manipulation statements have been verified to be safe, including complicated statements such as the pointer increment and the destructive update at line [8] of `join`.

2.1 Operational Semantic

We sketch an instrumented operational semantics for C that verifies the absence of out-of-bound violations while allowing pointer arithmetic, destructive updates and casting. The general idea is to define a non-standard low-level semantics that explicitly represents the base address of every memory location and the allocated size starting from the base address. This semantics is rigorous. It forbids programs with undefined ANSI-C behavior, but it also checks additional requirements reflecting good programming styles, such as dereferences beyond the null-termination byte. This



```
   void remove_newline (char* s) {
[1]       char *temp = strrchr (s, '\n');
[2]       if (temp == NULL)
[3]          exit (1);
[4]       *temp = 0;
   }

   void join (char *cp) {
          char temp[BUFSIZ], *tp;
[5]       if (!fgets (temp, BUFSIZ, stdin))
[6]          return;
[7]       remove_newline (temp);
[8]       *cp++ = ' ';
[9]       for (tp = temp; *tp == ' '; ++tp) ;
[10]      strcpy (cp, tp);
   }

   void main () {
          char buf[BUFSIZ];
          char *cp;
[11]      while (fgets (buf, BUFSIZ, stdin)) {
[12]         remove_newline (buf);
[13]         for (cp = buf; *cp ; ++cp) ;
[14]         while (*--cp == ' ' ) ;
[15]         while (*cp == '.' )
[16]              join (cp + 1);
          }
   }
```

Figure 2.1: a string manipulation program based on the code of `fixwrites`.

semantics provides the foundation of the abstract interpretation conducted by the analysis, i.e., the abstract interpretation conservatively represents the states of this semantics. In addition, the string analysis statically verifies the absence of string errors by conservatively checking the preconditions of this semantics.

Definition 2.1.1 A **concrete state at a procedure** P is a tuple:

$$\mathit{state}^\natural = (\mathcal{L}^\natural, \mathcal{BA}^\natural, \mathit{aSize}^\natural, \mathit{loc}^\natural, \mathit{st}^\natural, \mathit{numBytes}^\natural, base^\natural)$$

where:

- $\mathcal{L}^\natural$ 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 $\mathcal{L}^\natural$.
- $\mathit{aSize}^\natural \colon \mathcal{BA}^\natural \to \mathbb{N}$ defines the allocation size in bytes of the memory region starting at a base address.
- $\mathit{loc}^\natural \colon \mathit{visvar}_P \to BA^\natural$ maps visible variables into their assigned global or stack locations (which is always a base address).
- $\mathit{st}^\natural \colon \mathcal{L}^\natural \to val$ defines the memory content, where

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

  is the set of possible values. The value $uninit$ represents uninitialized values; $undefined$ represents results from illegal memory access; *primitive refers to the set of C primitive type (*`char`, `int`, *etc.) values.*
- $\mathit{numBytes}^\natural \colon \mathcal{L}^\natural \to \mathbb{N}$ *defines for each location the number of bytes of the value stored starting at the location.*
- $base^\natural \colon \mathcal{L}^\natural \to \mathcal{BA}^\natural$ *maps every location to its base address.*

Intuitively, the state keeps track of the set of allocated locations ($\mathcal{L}^\natural$). The origin location of each memory region that is guaranteed to be contiguous is in $\mathcal{BA}^\natural$. In order to handle a destructive update to a variable via the address-of operation, $\mathit{loc}^\natural$ represents the address of variables, and $\mathit{st}^\natural$ maps locations into their values.

The reader is referred to [13, 11] for further elaboration on this semantics.

Attribute	Type	Intended Meaning
E.base	$\mathcal{BA}^\natural$	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 2.1: Sugared expression over the instrumental semantics. E_i's are pointer expressions without side effects.

2.2 Adding Safety Precondition

The first step in our string algorithm is a source to source transformation to include runtime checking for array bounds and string violations. This is concluded by adding `assert` statements over our instrumental semantics of a precondition that must hold prior to statement execution. For clarity, sugared expression as defined in Table 2.1 are used.

For every C expression, there is a condition that verifies the validity of the expression. Table 2.2 lists the generated assert expressions. On every dereference to an address, a check that the address is within bounds is generated. The upper bound is checked depending on whether the buffer is null-terminated. If it is, the dereferenced location is checked to be at or before the null-termination byte. For pointer arithmetic, the generated assert statement checks the requirement that the resultant reference is within bounds of the buffer. Similarly, for each library function, the user of our tool must provide a contract defining the function's precondition that must hold and a stub implementation describing the function's affect. For example, the precondition of `strcpy(char *dst, const char *src)` is

src.string && dst.alloc $>$ src.strlen && is_within_bounds(dst) && !is_overlap(dst,src)

2.3 Pointer Analysis

The next step of our analysis is a whole-program flow-insensitive pointer analysis designed to detect statically which pointers may point to the same base address. In particular, for every function, it provides a summary of all of its calling contexts. 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, the whole-program analysis of `remove_newline` see Fig. 3.2(a) yields that `s` (represented by node labeled $lv_{\text{s}}^{\text{remove\_newline}}$) may point to either the local `buf` or `temp` arrays (represented by $lv_{\text{buf}}^{\text{main}}$ and $lv_{\text{temp}}^{\text{join}}$). Conservatively analyzing the

C Exp.	Generated Precondition
$*E$	E.offset ≥ 0 && ((E.string && E.strlen ≥ 0) || (!E.string && E.alloc > 0))
E	$0 \geq E$.offset $< E$.alloc
$E + N$	$(0 \geq N \leq E$.alloc) || $(0 < N \geq E$.offset)

Table 2.2: Asserted preconditions for C expressions. E and N are pointer and integer expression, respectively, without side effects.

function's body with the two calling contexts, requires treating updates to integer properties as weak updates, for example the null-termination property due to execution of `*temp = 0`. Therefore, the analysis will fail to infer that the resulting pointer points-to a string. As a result, a false alarm will be issued. Our interprocedural analysis computes the effect of a procedure in a parameterized manner which enables, in many cases, to regard destructive updates as strong updates.

For every call site a *call points-to graphs*, CPT for short are computed. A CPT represents the possible pointer values that may arise at this call site. In each CPT each pointer argument is evaluated to point to a single node in a graph. In order to consider all possibilities, more than one CPT will be generated for a single call statement. The CPT for the calls to `remove_newline` at line [12] and at line [7] are shown in Fig. 3.2(b) and (c), respectively.

2.4 Static Integer Analysis

The analysis conservatively analyzes the generated program and computes an abstract representation for each program point representing all possible concrete states that may arise at this point for all execution paths. The abstract representation follows the concrete semantics and infers the interesting numeric properties of base addresses. Since the number of base addresses is potentially infinite, potentially multiple base addresses are abstractly represented by a single abstract base address. The abstract representation used in iCSSV infers numeric relations among integer variables and the string, strlen and alloc properties of abstract base addresses.

In theory, any sound integer analysis can be used. Because many of the tracked semantic properties are external to the procedure, and sometimes even to the whole application, it is essential to track relationships between constraint variables and not just possible values. Furthermore, many of the conditions inferred involve three or more properties. Given that our goal is to generate as few false messages as possible, the algorithm applies the linear-relation analysis [6, 14] which discovers linear inequalities among numerical variables. This method identifies

linear inequalities of the form: $\Sigma_{i=1}^{n} c_i x_i + b \geq 0$, where x_i is an integer variable and c_i and b are constants. In our case, x_i are the integer variables and the numerical properties of abstract base addresses. Upon termination of the integer analysis, the information at every control-flow node conservatively represents the inequalities that are guaranteed to hold whenever the control reaches the respective point. The reader is referred to [6, 14, 12] for information about integer analysis.

During integer analysis, each `assert` statement is verified. This is carried out by checking if the asserted integer expression is implied by the linear inequalities that hold at the corresponding control-flow node. If the assertion cannot be verified, a counter-example is generated. The counter-example describes the values of the constraint variables where a string error in the C program may arise.

Fig. 4.2 demonstrates how the static integer-analysis algorithm identifies the error in the pointer arithmetic in line $[14]$ of `main`. The algorithm discovers that the inequalities shown in Fig. 4.2 (a) hold before the execution of line $[10]$, and that when the equality shown in Fig. 4.2 (b) holds, a violation of the pointer arithmetic precondition occurs.

2.4.1 Interprocedural Analysis

Handling procedure calls, one of the more complicated tasks in static analysis, strongly influences the scalability and the precision of the entire analysis. CSSV handles procedure calls by requiring the user to provide a contract for each procedure, allowing one to specify what to check and where. Contracts have another advantage as they make it easier to scale to large programs. However, in many cases, providing contracts is not a feasible task. In iCSSV the need to provide contracts is avoided by computing a combined summary information for each procedure. The summary information is computed for each possible CPT graph. We check whether a precomputed summary information can be used. The summary information contains the exit state as a linear relationship of the entry state.

During the intraprocedural analysis, when a function call is encountered, we first compute the CPT graphs for this call. We check if this procedure has been previously analyzed and if the summary analysis can be used. Informally, the summary analysis can be used if the CPT graphs are isomorphic and the existing entry state is a subset of the computed entry state. This algorithm enables us in may cases to avoid multiple analyses of the same function. In our example, each function is analyzed once. Thus, the summary information of `remove_newline` is used during the intraprocedural analysis of `join` and the summary of `join` is used during the analysis of the `while` loop in `main`.

Chapter 3

Converting C programs to procedural IP

In this section, we present an algorithm for converting C programs into a pointer-free integer programs. The resultant program is not equivalent to the original program. Instead we guarantee that any string violation in the original program leads to a violation of an integer assertion in the resultant program.

The main idea is a transformation that maps integer properties of interest to *constraint variables* and generates an integer program manipulating the constraint variables. The IP contains assertions representing safety conditions of the original C program. If our analysis concludes that an assert condition in the IP is always satisfied, then the algorithm guarantees that the equivalent string violation in the C program can never occur. This transformation process has to handle the following complications:

- Pointer expressions and specifically cases where a pointer may or may-not point to a specific location.
- Dynamic locations, and specifically allocations sites where more than one location can be allocated
- C expressions that are too complex to transform or are of no interest to the IP
- Handle procedure calls in cases where there is no IP for the called function.

We propose a solution that uses a flow-insensitive whole-program pointer information to compute abstract locations. This unifies the treatment of ordinary stack allocated buffers with the treatment of pointers to buffers and dynamically allocated buffers. The generated integer program is non-deterministic, supporting C expressions that are not translated to IP.

This chapter explains the transformation and demonstrates it by using our running example in Fig. 2.1.

3.1 Preliminaries

3.1.1 CoreC

CoreC is a subset of C with the following restrictions:

1. Control-flow statements are either `if`, `goto` , `break`, or `continue`;
2. expressions are side-effect free and cannot be nested;
3. all assignments are statements;
4. declarations do not have initializations;
5. address-of formal variables is not allowed.

An algorithm for transforming C programs to CoreC is presented in [31]. Given a C program, it generates an equivalent CoreC program by adding new temporaries. iCSSV is defined and implemented for CoreC. In the rest of this paper, CoreC is used instead of C.

3.1.2 Contracts

iCSSV uses contracts to simulate the behavior of procedures whose code is not available, e.g. library routines. Contracts describe their expected inputs, side-effects, and expected output. In this thesis, contracts are written in the style of Larch [21]. Our implementation actually supports a more general executable language similar to [25], which can include loops. Contracts are specified in the `.h` file. Every prototype declaration of a function *f* has the form:

$$\begin{array}{lll} \langle type \rangle\, f\,(\cdots) & \textbf{requires} & \langle e \rangle \\ & \textbf{modifies} & \langle e \rangle, \langle e \rangle, \ldots, \langle e \rangle \\ & \textbf{ensures} & \langle e \rangle; \end{array}$$

The contract defines the precondition required to hold whenever *f* is invoked, the side-effects of the function *f*, i.e., the objects that may be modified during invocations of *f*, and the postcondition that is guaranteed to hold on the modified objects. Here, $\langle e \rangle$ is a C expression, without function calls, over global variables and the formal parameters of *f*. We allow *attributes* of the form defined in Table 3.1 and displayed in Fig. 3.1. A designated variable `return_value` denotes the return value of *f*. The special syntax $\lceil \langle e \rangle \rceil_{pre}$ denotes the value of $\langle e \rangle$ when *f* is invoked. Although not required, the contract mechanism enables specification of pointer values. In addition, a shorthand expression `is_within_bounds(arg)` is allowed to indicate that `arg` points within the bounds of a buffer.
In our running example, since the code of *fgets*, *strcpy*, *strrchr* is not supplied, contracts will be

Attribute	Intended Meaning
exp.base	The base address of *exp*
exp.offset	The offset of *exp*, i.e., exp - exp.base
exp.is_nullt	Is *exp* pointing to a null-terminated string?
exp.strlen	The length of the string pointed-to by *exp*
exp.alloc	The number of bytes allocated from *exp*

Table 3.1: Attributes in the contract language.

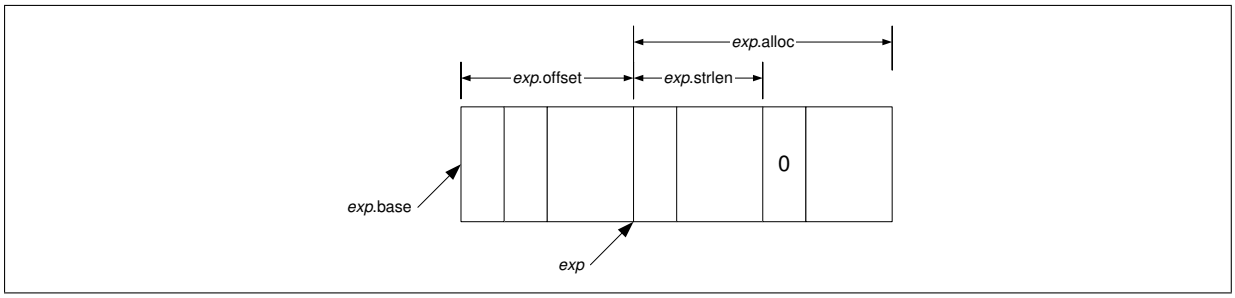


Figure 3.1: Graphical representation of the contract-language attributes.

used.
In iCSSV, the contracts are translated to CoreC and then to IP, and they are used as a replacement to the actual code of the functions with the contracts. Consequently, in the rest of the paper, functions having contracts instead of code will be treated exactly like functions with available code.

3.2 Call points-to graph

The first phase of the iCSSV algorithm computes an abstraction of all potential pointer relationships between locations in concrete states that may occur in each procedure entry site. For this computation the algorithm uses the global abstract points-to state of the entire program, which is a sound approximation of the pointer relationships in the entire program. Formally, the definition of the global points-to state is as follows:

Definition 3.2.1 The global abstract points-to state of the whole program, GPT for short, $Gstate = (\mathcal{BA}, loc, pt, sm)$ where:

- $\mathcal{BA}$ is a set of abstract locations.
- $loc\colon var \rightarrow 2^{\mathcal{BA}}$ maps a variable into a set of abstract locations representing the variable's global or stack locations.

- *pt*: $\mathcal{BA} \rightarrow 2^{\mathcal{BA}}$ abstracts the pointers from one location to another.
- *sm*: $\mathcal{BA} \rightarrow \{1, \infty\}$ abstracts the number of concrete base addresses represented by an abstract location. It represents summary information as discussed above.

A GPT summarizes locations into summary locations yielding a finite set of abstract locations. In addition, the $Gstate$ ensures us that if two pointers expressions, say e_1 and e_2 may point to the same concrete location then additional dereferences from e_1 and e_2 are represented by the same abstract locations, i.e., if p and q may point to the same location then $p \rightarrow n$ and $q \rightarrow n$ are represented by the same abstract location.

When analyzing a single procedure,referred to as P, called from a call site, referred to as C, only locations that can be accessed during the execution of P when it is called from C are of interest. Therefore, we define the notion of reachable locations.

Definition 3.2.2 In a concrete state, a location $l^\natural$ is reachable if there exists a visible variable whose store contents can (indirectly) include $l^\natural$ (i.e., there is an expression whose L-value is $l^\natural$).

Using the notion of reachable locations, we can define *call points-to graphs*.

Definition 3.2.3 A *call points-to state*, CPT for short, is a subset of the program's GPT, containing all the abstract locations reachable from a procedure's local variables and the program's global variables, as well as abstract locations reachable from both the formal parameters and corresponding actual arguments at the call site C.

A distinct CPT is associated with each possible combination of abstract locations associated with the actual arguments at the call site. A CPT represents all the potential pointer relationships between locations in concrete states that may occur during a specific call to that procedure. It is computed from the program's complete points-to state. For every call-site, a unique CPT is generated for each possible combination of abstract locations, representing memory locations pointed by the program pointers in the call's actual arguments. Therefore, each call site introduces one or many such states. This ensures us that our analysis operates on every possible case of pointer aliasing. Due to the finite number of abstract locations the number of CPT states is bounded.

During the translation of C programs to IP, iCSSV relies on the notion of points-to-isomorphism which is a strict isomorphism between points-to states.

Definition 3.2.4 A state isomorphism mapping *i* between two points-to states *G* and *G'* is a **points-to-isomorphism** iff all the following conditions are satisfied:

1. $var_G \equiv var_{G'}$

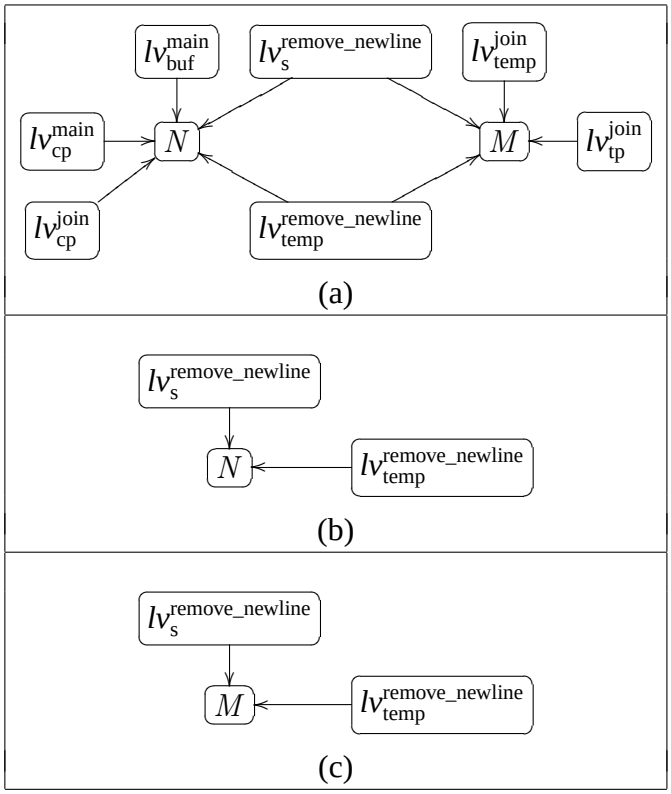


Figure 3.2: The whole-program points-to information for the running example (a), excluding pointers of the library procedures `strcpy`, `strrchr` and `fgets` and the CPT for the calls to `remove_newline` at line [12] (b) and at line [7]. Note that the two CPTs are points-to-isomorphic.

2. $\forall v \in var_G : loc_G(v) = loc_{G'}(v)$

For example, in Fig. 3.2 the two CPT graphs (b) and (c) which abstract the points-to graphs in the calls to remove_newline in lines $[12]$ and $[7]$ respectively, are points-to isomorphic.

3.3 C2IP

In order to apply the interprocedural integer analysis, iCSSV transforms a C procedure with a specific CPT to an integer program (IP for short). The generated IP tracks the string and integer manipulations of the program. The IP is nondeterministic, reflecting the fact that not all values are known. The symbol `unknown`stands for an undetermined value. This value can be assigned and used as a branch condition.
The semantics of the **assume** construct in the IP is to restrict the behavior of nondeterministic programs. Finally, for clarity, mathematical constructs are used in the IP. The IP includes *constraint variables* used to denote interesting semantic properties of *abstract base location*. An abstract base location represents potentially many concrete base locations. For example, a constraint variable $rv_{buf}.aSize$ represents the allocation size of the base address which is the stack location of `buf`. The translation to IP generates update statements assigning new values to constraint variables, reflecting the changes in the semantic properties. **Assert** statements over the constraint variables are generated for checking the safety of basic C expressions. The constraint variables iCSSV tracks are:

- $l.val$ to represent potential primitive values stored in the locations represented by l.
- $l.offset$ to represent potential offsets of the pointers represented by l, i.e. $l.offset$ conservatively represents $index^{\natural}(st^{\natural}(l^{\natural}))$ for every location $l^{\natural}$ represented by l.
- $l.aSize$, $l.is\_nullt$, $l.len$ to describe the allocation size, whether the base address contains a null terminated string, and the length of the string (excluding the null byte) of all locations represented by l.

Safety checks Transforming C expressions to IP statements involves querying the CPT to obtain the abstract location a pointer *may* point to. For simplicity, we will now assume that every pointer may point to only a single non-summary abstract location. Table 3.2 shows the translated asserts. We denote by lv_x (rv_x) the resulting abstract location representing the L-value (R-value) of expression x. iCSSV handling of arbitrary CPT graphs is explained later in this section. For each C statement, iCSSV generates conditions validating the safety of the statement. For example, for every address dereference iCSSV generates a check that the address

C Exp.	Generated IP Condition
$*p$	$lv_{\mathrm{p}}.offset \geq 0 \wedge$ $((rv_{\mathrm{p}}.is\_nullt \wedge lv_{\mathrm{p}}.offset \leq rv_{\mathrm{p}}.len) \vee$ $(\neg rv_{\mathrm{p}}.is\_nullt) \wedge lv_{\mathrm{p}}.offset < rv_{\mathrm{p}}.aSize))$
$p + i$	$lv_{\mathrm{p}}.offset + lv_{\mathrm{i}}.val \geq 0 \wedge$ $lv_{\mathrm{p}}.offset + lv_{\mathrm{i}}.val \leq rv_{\mathrm{p}}.aSize$

Table 3.2: Generated safety conditions

is within bounds. The upper bound used in this check, depends on whether the buffer contains a null-terminated string or not.

C statements The IP program also includes update statements to reflect semantic changes regarding the properties tracked. Table 3.3 displays the generated statements from transforming CoreC statements and conditional expressions involving pointers to buffers, which is the interesting part of the translation.

On allocation, the resultant pointer always points to a base address. Therefore, its offset is always zero. We set the allocation size of the abstract location that represents the newly allocated location. Destructive updates are separated into two cases: (i) The assignment of the null character, which sets the buffer to a null-terminated string. The length of the string is the location of the first zero byte. C2IP generates a check that all dereferences reference bytes preceding the null-termination byte (if it exists). We can therefore safely assume that when assigning a null-termination byte it is the first null value in the buffer. (ii) In the assignment of a non-zero character, it is checked whether an existing null-termination byte is overwritten.

To increase precision, certain program conditions are interpreted. The second part of Table 3.3 shows the interpreted conditions. When checking for null-termination, C2IP replaces the original condition with a condition over constraint variables that track the existence of a null-character and the string's length. Pointer comparisons are replaced by expressions over the appropriate `offset` constraint variables.

The third part of Table 3.3 shows how procedure calls and returns are translated. When translating calls, iCSSV replaces pointers with pointer-offsets. For every abstract location visible to both the calling and the called procedure, the generated call passes the constraint variables associated with the abstract location as parameters to the called procedure. These parameters are labelled "in out", meaning that upon return from the called procedure, the values of the actual arguments are updated to the values of the matching formal parameters. This behavior soundly simulates cross-procedural updates through pointers.

Return statements are translated as assignments to a variable with a unique name. The successor instruction to this assignment is the procedure's exit node. When the return value is

C Construct	IP Statements
p = Alloc(i);	$lv_{\text{p}}.offset := 0$; $rv_{\text{p}}.aSize := lv_{\text{i}}.val$; $rv_{\text{p}}.is\_nullt := false$;
p = q + i;	$lv_{\text{p}}.offset := lv_{\text{q}}.offset + lv_{\text{i}}.val$;

*p = c;	if $c = 0$ {
	$rv_{\text{p}}.len := lv_{\text{p}}.offset$;
	$rv_{\text{p}}.is\_nullt := true$; }
	else
	if $rv_{\text{p}}.is\_nullt \wedge lv_{\text{p}}.offset = rv_{\text{p}}.len$
	$lv_{\text{p}}.is\_nullt :=$ `unknown`;

c = *p;	if $rv_{\text{p}}.is\_nullt \wedge lv_{\text{p}}.offset = rv_{\text{p}}.len$ $lv_{\text{c}}.val := 0$; else $lv_{\text{c}}.val :=$ `unknown`;
*p == 0	$rv_{\text{p}}.is\_nullt \wedge rv_{\text{p}}.len = lv_{\text{p}}.offset$
p > q	$lv_{\text{p}}.offset > lv_{\text{q}}.offset$
$f(v1, \ldots, vm,$ $p1, \ldots, pn)$;	$f(lv_{\text{v1}}.val, \ldots, lv_{\text{vm}}.val,$ $lv_{\text{p1}}.offset, \ldots, lv_{\text{pn}}.offset,$ $\ldots rv_{\text{li}}.aSize, rv_{\text{li}}.is\_nullt, rv_{\text{li}}.len \ldots)$;
return p;	$lv_{\text{\_return\_val}} := lv_{\text{p}}.offset$;
return i;	$lv_{\text{\_return\_val}} := lv_{\text{i}}.val$;
p.alloc	$rv_{\text{p}}.aSize - lv_{\text{p}}.offset$
p.offset	$lv_{\text{p}}.offset$
p.is_nullt	$rv_{\text{p}}.is\_nullt$
p.strlen	$rv_{\text{p}}.len - lv_{\text{p}}.offset$

Table 3.3: The generated transformation for C statements, conditional expressions and for contract-language attributes. `p` and `q` are variables of type pointer to char. `i` and `c` are variables of int type. `Alloc` is a memory allocation routine, e.g., `malloc` and `alloca`.

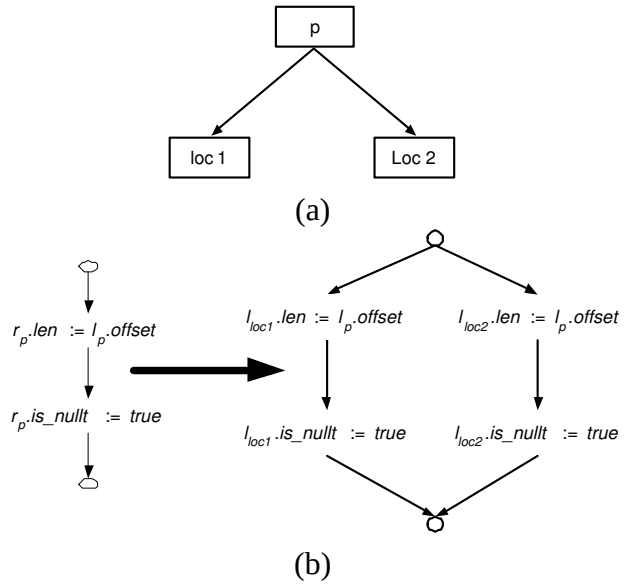


Figure 3.3: Translating the C statement *$p = 0$. In the case where p is associated with the abstract points-to set $\{loc1, loc2\}$ (fig. a) the IP instructions are duplicated. Note that the two generated paths merge only at the end of the C statement and not after every IP instruction, thus achieving better accuracy.

a pointer, our analysis assigns the constraint variable abstracting the pointer offset to the return value variable. The justification for this behavior is that the abstract locations that represent the target of this pointer are shared between the calling procedure and the called one. Therefore, all the constraint variables associated with these locations would be updated upon return through the use of appropriate "in out" parameters. The offset, however, is the only property associated with the pointer, and therefore should be updated through the return statement.

From pointers to integers C2IP uses CPT graphs to find the suitable constraint variable for each statement concerning pointers in the C program. For each statement, the tables in the following paragraphs define which constraint variables must be checked or modified to ensure the safety of the statement and to simulate the program run. The CPT is used to find which abstract locations are manipulated in every statement in order to generate IP statements with the suitable constraint variables.

In the case where an L-value (R-value) in the abstract points-to state includes more than one abstract location or a summary abstract location, the translation rules of Table 3.3 need to be changed to guarantee sound results. To reflect the fact that a base address represented by l may or may not be modified, C2IP generates every statement (shown in Table 3.3) to operate on every abstract location in the abstract points-to set. For a summary abstract location, the analysis translates the statement as a nondeterministic assignment, under an `if (unknown)` statement. In addition, the analysis must take into account all possible values of a pointer, and

verify expressions on all possible pointer values. This applies to all generated assert statements and program conditions. The translation of C statement referencing a pointer associated with an abstract points-to set with several abstract locations, is demonstrated in Fig. 3.3.

For each procedure iCSSV partitions the set of CPT graphs associated with calls to that procedure to equivalency classes defined by points-to-isomorphism. iCSSV then translates the procedure from C to IP once for each such equivalency class, using one of the CPT graphs of that class to find the L-values and R-values of the pointers in the procedure. In the running example, the two calls to remove_newline are associated with points-to isomorphic CPT graphs. Therefore, the function is translated only once. Without losing generality, we assume that it is translated with the CPT associated with the call at line $[12]$. A call to that clone will be generated in both line $[7]$ and $[12]$. However, the call at line $[12]$ is associated with the identity points-to isomorphism, while the call at line $[7]$ is associated with the points-to isomorphism mapping abstract location M to abstract location N. The isomorphism is used during the integer analysis to update the constraint variables associated with the abstract location when `remove_newline()` returns.

The Complexity of C2IP For each procedure, the number of constraint variables in the IP is $O(V)$ where V is the number of variables and allocation sites in the C program. Because a pointer may point to V abstract locations, the translation of a C expression that contains one pointer generates $O(V)$ IP statements. Therefore, the size of the IP is $O(S * V)$, where S is the number of C expressions in the procedure. This is an order-of-magnitude improvement over the transformation in [12], which generates $O(V^2)$ variables and $O(S * V^2)$ statements.

To calculate the complexity of the IP of the entire program, this number is multiplied by the number P of procedures in the code. Since each procedure is cloned according to the number of non-isomorphic CPT graphs associated with calls to it, this complexity is multiplied by 2^V. This stems from the fact that each combination of abstract locations in the procedure's actual arguments may result in a new CPT graph, not isomorphic to any existing graph. Therefore, the complexity of the entire IP is $O(P*2^V*V*S)$ statements. However, in practice, because of CPT isomorphisms, the number of cloned routines is $O(P)$, resulting in the IP having $O(P * V * S)$ statements.

Chapter 4

Interprocedural analysis of a procedural IP

4.1 The abstraction

CSSV analyzes the IP and reports potential assert violations. In theory, any sound integer abstraction can be used. Because many of the tracked semantic properties are external to the procedure, and sometimes even to the entire application, it is essential to track relationships between constraint variables and not just possible values. Furthermore, many of the inferred conditions involve three and more properties, e.g., dereference safety checks. Given that our goal is to generate as few false alarms as possible, iCSSV applies the linear-relation analysis [6, 14] which discovers linear inequalities among numerical variables. This method identifies linear inequalities of the form: $\Sigma_{i=1}^{n} c_i x_i + b \geq 0$, where x_i is an integer variable and c_i and b are constants. In our case, x_i are the constraint variables. Upon termination of the integer analysis, the information at every control-flow node conservatively represents the inequalities that are guaranteed to hold whenever the control reaches the respective point. The reader is referred to [6, 14, 12] for information about integer analysis.

A unique aspect of our usage of the linear-relation analysis is the use of a 2-vocabulary. To summarize the effects of a procedure, its side effects and outputs are required to be expressed as a function of the procedure's inputs. 2-vocabulary is the device through which this goal is achieved. A 2-vocabulary polyhedron is a polyhedron that has two dimensions associated with each IP variable v visible in a procedure P — one dimension is labelled $v[in]$ and the other $v[out]$. $v[in]$ represents the possible values to the variable v at the entry to the procedure, and $v[out]$ represents the possible values to v during and at the end of the execution of the procedure. The analysis will express the variables in $V_P[out]$ as a function of variables $V_P[in]$.

4.2 Interprocedural linear-relation analysis

Pseudo code for the interprocedural linear-relation analysis is given in Fig. 4.1. It is used throughout this section.

The algorithm is a work-list algorithm. The work-list used here contains control-flow graph nodes and the abstract value calculated for that node during the analysis. The work-list is initialized in line $[4]$ with the start node of $Main$, the analyzed program's entry point. This node is associated with the abstract value $\top$, reflecting the fact that at the beginning of the analysis, there is no knowledge about the state of the program.

Other data structures used are a call stack, $CallStack$, which stores the call nodes in a stack structure. The stack is used so that the solution would be over valid paths only. Another structure, also based on a stack , is $ValuesForMeet$. This stack holds the abstract values representing the call context at each call site. The usage of the abstract values of this stack will be discussed in Section 4.2.2.

4.2.1 Processing procedure calls

Our analysis uses linear relations to express the relationships between the state at the return site and the state at the call site. As previously discussed, this summarization of a procedure's effects is achieved by using a 2-vocabulary representation. At call sites, the analysis needs to set up the input variables to the called procedure P, i.e. $V_P[in]$. This task is performed by the $BuildContext$ routine. The input to this routine is v, the input abstract value to the call site. For every variable in V_P, two dimensions are added to v, generating a 2-vocabulary for V_P (lines $[43]$-$[45]$). After that, the context of the call is initialized by performing for each formal parameter f, an assignment where $f[in]$ is assigned the value of the corresponding actual argument (lines $[46]$-$[49]$). Afterwards, $f[out]$ is assigned the value of $f[in]$ (line $[51]$-$[53]$), generating the initial connection between the context at the start of the called procedure and the state during and after its execution. From this point until returning from the called procedure, all of the operations are performed on out variables. After all of the dimensions associated with the variables of P have been initialized, the dimensions associated with variables of the called procedure are eliminated from the abstract value (line $[54]$). This transformation retains only the context of the call in the abstract value.

After the specific call context has been calculated, the algorithm tries to use previous calculations of the called procedure in order to save analysis time and space. In order to use a previous call to the procedure as a summary, the context of that previous call must contain the current context. If so, the analysis will insert the return node along with the previous exit value to the work list. If the context of the previous call does not contain the current, a join between the two contexts is performed, so that the chances for summarizing the procedure in the next call will increase.

```
   algorithm Analyze(G^#_IP)
   begin
[1]  for each p ∈ Call_p do
[2]    Context(p) := ⊥
[3]  od
[4]  WorkList := {⟨s_main, ⊤⟩}
[5]  CallStack := ∅
[6]  ValuesForMeet := ∅
[7]  while WorkList ≠ ∅ do
[8]    Select and remove a node ⟨n, v⟩ from WorkList
[9]    switch n
[10]     case n ∈ Call_P :
[11]       context_n = BuildContext(n, P, v)
[12]       AllContexts_P := Context(P)
[13]       if context_n ⊑ AllContexts_P then
[14]         Insert ⟨returnSite(n), exitVal(P)⟩ to WorkList
[15]       else
[16]         Context(P) := AllContexts_P ⊔ context_n
[17]         Insert ⟨s_P, Context(P)⟩ to WorkList
[18]         push n to CallStack
[19]       fi
[20]     end case

[21]     case n = Exit_P
[22]       exitVal(P) := v
[23]       n_call := top CallStack
[24]       CallStack pop
[25]       Insert ⟨returnSite(n_call), v⟩ to WorkList
[26]     end case

[27]     case n ∈ Return_P
[28]       v_ret = PerformMeet(n, P, v)
[29]       for each n' ∈ N such that (n, n') ∈ E do
[30]         if n' is in WorkList then
[31]           remove ⟨n', v'⟩ from WorkList
[32]           Insert ⟨n', v' ⊔ v_ret⟩ to WorkList
[33]         else
[34]           Insert ⟨n', v_ret⟩ to Worklist
[35]         fi
[36]       od
[37]     end case

[38]     otherwise
[39]       Perform the integer operation in node n. Update
           successors and perform widenings on back-edges
[40]     end case
[41]   end switch
[42] od
   end
```

```
   procedure BuildContext(n, P, v)
   begin
[43] for each var ∈ visvars_P do
[44]    Add two dimensions and embed to polyhedron v.
        Label one var[in] and the other var[out].
[45] od
[46] for each f_i ∈ formals_P do
[47]    a_i := actual argument matching f_i
[48]    v := affine_image(v, f_i[in] = a_i)
[49] od
[50] push v in ValuesForMeet
[51] for each f_i ∈ formals_P do
[52]    v = affine_image(v, f_i[out] = f_i[in])
[53] od
[54] Remove from v all dimensions corresponding
     to V_procOfn
[55] return v
   end

   procedure PerformMeet(n, P, v_e)
   begin
[56] v_call = top ValuesForMeet
[57] pop ValuesForMeet
[58] Add dimensions to v corresponding with
     V_procOf(n) and embed the old dimensions.
[59] v = v ⊓ v_call
[60] for each f in formals_P do
[61]    if out ∈ mode(f) then
[62]       a := matching actual argument
[63]       v = affine_image(v, a = f[out])
[64]    fi
[65] od
[66] Remove from v all dimensions corresponding with V_P
[67] return v
   end
```

Figure 4.1: The interprocedural integer analysis algorithm

4.2.2 Returning from a procedure

When the algorithm returns from a procedure, it performs two tasks:

1. Makes the returned abstract value specific to the call site — using a meet ($\sqcap$) operation.

2. Updates the state according to the returned information.

Due to the fact that the algorithm may have used a summary in the matching procedure call, in order to increase precision, the algorithm performs a meet between the abstract value returned from the called procedure, i.e. $exit_val$ and the abstract value at the call site, i.e. v. Technically, in order to perform this task, the dimensions of the two polyhedrons need to be matched. Therefore the meet is performed on the v_{call} abstract value instead of v. v_{call} is similar to v, but it contains dimensions associated with the variables of the called procedures P. The out variables of P are unconstrained in V_{call}. The $exit_val$ polyhedron is added unconstrained dimensions before the actual meet, representing the variables of the calling procedure. Therefore, the only dimensions that are constrained in both polyhedrons are the ones which represent the in variables of V_P, i.e. the call context. Furthermore, in V_{call} these variables hold the possible values at the specific call site, whereas in $exit_val$ they are associated with all possible values in all call sites to P. The meet operation will therefore make the dimensions associated with $V_P[out]$ contain only values specific to the call site.

After the meet is performed, for every formal parameter which is labelled as an "out" parameter, the algorithm updates the appropriate actual argument with the value of the formal parameter at the return site. This is used to simulate an update of shared data between the calling and called procedure through pointers.

4.2.3 Assert Checking

During integer analysis, each `assert` statement is verified. This is performed by checking whether the asserted integer expression is implied by the linear inequalities that hold at the corresponding control-flow node. If the assertion cannot be verified, a counter-example is generated. The counter-example describes the values of the constraint variables where a string error in the C program may arise.

Fig. 4.2 demonstrates how the static integer-analysis algorithm identifies the error in the while loop in line $[14]$ of `main`. The algorithm discovers that the inequalities shown in Fig. 4.2 (a) hold before and during the execution of line $[14]$, causing a violation of the dereference safety condition when the equality in Fig. 4.2 (b) holds.

$$
\begin{array}{lcl}
rv_{\text{buf}}.aSize & = & BUFSIZE \\
rv_{\text{buf}}.is\_nullt & = & true \\
rv_{\text{buf}}.len & \geq & 1 \\
rv_{\text{buf}}.aSize & \geq & rv_{\text{buf}}.len + 2 \\
rv_{\text{buf}}.len & \geq & lv_{\text{cp}}.offset
\end{array}
$$

(a)

[14] while(*–cp == ' ');
require($lv_{\text{s}}.offset \geq 0$)
`error: the requirement may be violated when:`

$$lv_{\text{cp}} \leq 0$$

(b)

Figure 4.2: A report on the error in line [14] of `main`. The derived inequalities before execution of line [14] of `main` (a), and a counter example (b).

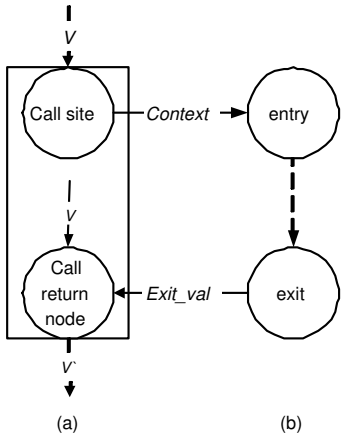


Figure 4.3: Flow of abstract values in interprocedural analysis from the caller on the left (a) to the called procedure on the right (b).

4.3 Optimizations

4.3.1 Calculating the *Use* & *Mod* sets

The most significant factor effecting efficiency, in terms of both computation time and space, is the space dimension of the polyhedron. An optimization to the basic algorithm that reduces the space dimension of the polyhedrons, is presented here. The optimization is to precede the algorithm by a simple, linear analysis that conservatively determines which variables are needed to be associated with dimensions in the polyhedron, and which are not.

Definition 4.3.1 A function's *Use* set contains all of the variables and properties of abstract locations *used* by that function, or by a function that is called by it.

Definition 4.3.2 A function's *Mod* set contains all of the variables and properties of abstract locations *modified* by that function, or by a function that is called by it.

Calculating the *Use* & *Mod* sets of each function involves a recursive traversal of the call graph and combining the *Use* & *Mod* sets of each function with those of the functions called by it. After these sets are calculated, they are used to decrease the dimensions of the polyhedrons while still preserving soundness. A variable which is not used by a procedure or by any procedure called by it, i.e., is not in the *use* set, does not need to be associated with a dimension of a polyhedron in the analysis of that procedure. Similarly, for a variable which is not in the *Mod* set of a function, the analysis does not need to use a 2-vocabulary — one dimension associated with this variable will suffice.

Chapter 5

Empirical Results

We implemented a prototype of iCSSV. The implementation relies on the CoreC translator developed by Greta Yorsh at Tel-Aviv University. This tool and our CoreC to IP translator are built upon the AST-toolkit developed at Microsoft Research. iCSSV 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. Partial must information on pointer aliases is extracted from these edges. The integer analysis was implemented using the Parma Polyhedra library, developed at the University of Parma.

Due to fact that the most dominant factor effecting computation time and space is the space dimension of the polyhedrons during the analysis, several optimizations to the basic algorithm were made in the implementation of iCSSV. Liveness analysis was added because the translation to CoreC introduces a very large number of temporary variables. Another optimization resulted from an analysis of some early results, which identified that in many cases several dimensions of the polyhedrons hold a single constant value, i.e. the dimension x_i is constrained to some constant C: $x_i = C$, and not, for example, $x_i \leq C$. Due to this fact, iCSSV uses a cross-domain of constants and polyhedrons, adding dimensions to the polyhedrons only when a variable can be assigned more than one value.

We ran our implementation on the following program:

1. fixwrites - a TEXutility with numerous string manipulations.

2. apr_getpass - a password retrieving library routine from the apr libraries, part of the apache web-server.

We used both the polyhedra abstract domain and the octagons domain in our analysis. Octagons are polyhedrons that contain only constraints of the form $\pm x \pm y \geq c$. Other constraints have been discarded. This behavior is sound, yet inaccurate. The results are in Table 5.2.

Program	Procedures	loc	Errors	Polyhedra			
				FA	Time	Space	summary
fixwrites	12	424	2	143	1190s	206MB	51.8%
apr_getpass	7	277	0	23	2.0s	9MB	48%

Table 5.1: Test results using the Polyhedra abstract domains.

Program	Procedures	loc	Errors	Octagons			
				FA	Time	Space	summary
fixwrites	12	424	2	175	226.1s	44MB	77.5%
apr_getpass	7	277	0	22	0.7s	12MB	48%

Table 5.2: Test results using the Octagons abstract domains.

5.1 False alarms

As evident from the empirical results, many false alarms were reported by iCSSV on the two test-programs. The major sources of the false alarms are:

1. Inexact simulation of the contents of the string.
2. Unlimited widenings.
3. Information loss during join operations on convex polyhedra.

To explain these, I will portray them on a simple implementation of strlen, which is portrayed in Fig. 5.1 along with its translation to CoreC. The translation of CoreC to integer program is in Fig. 5.2. Let us assume that the algorithm is analyzing a call to strlen with a string parameter of length 10 in a buffer of size 15. The polyhedron representing the context to the call will be:

$$s.offset = 0$$
$$L.aSize = 15$$
$$L.is\_nullt = 1$$
$$L.len = 10$$

For simplicity, I leave out the details of the 2-vocabulary representation for the analysis of this example. At the end of the strlen, the __return_val variable should be 10, representing the exact return value in the given context, and no assertion errors should be reported. However, our analysis fails to realize this, failing the assertions and assigning __return_val with all values between 0 and 14. Each one of the following inaccuracies can alone cause this behavior.

5.1.1 Inexact simulation of the string's contents

Lines $[6]-[9]$ are the translation of the c = *src; statement in the CoreC program. The translation here is not accurate enough — if the offset of s is not equal to the string's length, c is assigned

```
int strlen(const char* src)
{
  int len;
  while(*src) {
    src++;
    len++;
  }
  return len;
}
```

(a)

```
int strlen(const char* src)
{
  int len;
  char c
label1:
  c = *src;
  if(c != 0) {
    src = src + 1;
    len = len + 1;
    goto label1;
  }
  return len;
}
```

(b)

Figure 5.1: An implementation of strlen (a) and its translation to CoreC (b).

```
      istrlen(in s.offset,
              in out L.aSize, in out L.is_nullt, in out L.len)
      {
[1]     len := 0;
[2]label1:
[3]     assert(s.offset ≥ 0 ∧
[4]             ((L.is_nullt ∧ s.offset ≤ L.len) ∨
[5]             ((¬L.is_nullt) ∧ s.offset < L.aSize)));
[6]     if(L.is_nullt ∧ s.offset = L.len)
[7]       c := 0;
[8]     else
[9]       c := unknown;
[10]    if (unknown) {
[11]      assume(c != 0);
[12]      assert(s.offset + 1 geq 0 ∧
[13]               s.offset + 1 ≤ L.aSize);
[14]      s.offset := s.offset + 1;
[15]      len := len + 1;
[16]      goto label1;
[17]    }
[18]    else {
[19]      assume(c = 0);
[20]    }
[21]    __return_val := len;
      }
```

Figure 5.2: The result of the translation of the CoreC version of strlen to IP. L is an abstract location pointed-to by s.

the value `unknown`($\top$). The effect of this assignment is that in line [19], the assume(c = 0) statement results in the following non empty polyhedron:

$$s.offset = 0$$
$$L.aSize = 15$$
$$L.is\_nullt = 1$$
$$L.len = 10$$
$$c = 0$$
$$len = 0$$

Consequently, in line [21] the assignment to $\_\_return\_val$ will add the value 0 to the possible values of this variable. Due to the fact that the assignment to c is performed in a loop on several values of the offset of s, and thus also on several values of len, this erroneous behavior will repeat itself for all of the possible values of $s.offset/len$. Therefore each value of len that is in the polyhedron in line [8], will be assigned later to $\_\_return\_val$. The next inaccuracy concerning the widening operation manages to make all values of len possible in line [8] (except for values outside the buffer, which are eliminated by the safety check of line [5]).

In order to solve this problem, the translation of statements of the from `c = *src;` must be changed. My suggestion is to add another case, translating the statement to:

if $rv_{\text{p}}.is\_nullt \wedge lv_{\text{p}}.offset = rv_{\text{p}}.len$
  $lv_{\text{c}}.val := 0;$
else {
  $lv_{\text{c}}.val :=$ `unknown`;
  if $rv_{\text{p}}.is\_nullt \wedge lv_{\text{p}}.offset = rv_{\text{p}}.len$
    assume($c \geq 1$);
}

The addition ensures us that if R-value of p is a null-terminated string and p points inside the string, i.e. before the null character, c will be higher than 0. This translation is sound and yet more accurate than the current one. It will ensure that c is definitely 0 only when the offset of p is equal to the length of the string, and possibly 0 only after the first null character.

5.1.2 Unlimited widenings

the `goto` command in line [16] results in a back-edge to line [2]. Widening operations are made on back-edges and in this case, the widening will result in the constraint on the offset $s.offset \geq 0$. This will result in an assert violation, and after performing error-recovery, the constraints on the offset will be $0 \leq s.offset \geq 14$ (recall that in our example, $L.aSize$, the allocation size of the buffer, is 15, therefore, in order to satisfy the assertion condition, $s.offset$

is less than 15). Combined with the previous inaccuracy concerning the contents of the string, this will result in the constraint $0 \leq \_\_return\_val \leq 14$ in the exit value of strlen.

The origin of this false alarm and inaccuracy in the return-value is the fact that the performed widening is unlimited, meaning that the widened dimensions are not bounded. Using loop invariants, the widened dimensions can be bounded. For example, in the loop `while(*src) { ... src++;}`, if the condition $rv_{\text{src}}.is\_nullt \wedge lv_{\text{src}}.offset \leq rv_{\text{src}}.len$ is satisfied in an iteration, then it will also be satisfied before the next iteration. This observation can be used in the analyzer to bound the offset, using the notion of widening with constraint implemented in the Parma library. The Parma library implements a widening operation where if a set of constraints is satisfied by both arguments to the widening, the widening will return a polyhedron which satisfies the given constraints. The conditionality of the bounded widening makes the analysis sound in cases where the loop does not satisfy the constraints, i.e. writes the null character or points beyond the null. In these cases the widening will not use the constraints and the widened dimensions will be unbounded.

The challenging problem here is therefore to automatically find during translation the set of constraints that can be used in the limited widening operation.

5.1.3 Information loss in the join operation

Even if the two suggested changes above are implemented, there will still be an inaccuracy in the return value of the function. This happens due to the loss of information caused by the join operation on convex polyhedra. Let us assume that the two changes above have been implemented, i.e. the analysis now differentiates between the contents before the null character and after it, and the widening is now bounded. Because of these changes, the offset of s would be, in our example, between 0 and 10, and the corresponding polyhedron:

$$
\begin{gathered}
0 \leq s.offset \leq 10 \\
L.aSize = 15 \\
L.is\_nullt = 1 \\
L.len = 10 \\
0 \leq len \leq 10 \\
c = unknown
\end{gathered}
$$

Due to the more exact analysis of the string contents we would get the following polyhedrons. For the case where $s.offset = L.len$:

$$
\begin{gathered}
s.offset = 10 \\
L.aSize = 15 \\
L.is\_nullt = 1 \\
L.len = 10
\end{gathered}
$$

$$len = 10$$
$$c = 0$$

For the case where the offset is lower than the length:

$$0 \leq s.offset \leq 9$$
$$L.aSize = 15$$
$$L.is\_nullt = 1$$
$$L.len = 10$$
$$0 \leq len \leq 9$$
$$c \geq 1$$

Because the widening is limited, the case where the offset is larger then the length of the string is non-existent in the portrayed call context. At the end of the if …then …else … clause the two abstract values will be joined, in our case a poly-hull operation will be performed. This will result in the following polyhedron:

$$0 \leq s.offset \leq 10$$
$$L.aSize = 15$$
$$L.is\_nullt = 1$$
$$L.len = 10$$
$$0 \leq len \leq 10$$
$$c \geq 0$$

In the above polyhedron, the two separate cases have been joined, and the information that the contents of the string preceding the null character must be greater than 0, was lost. As previously explained, this will result in the values of 0 to 9 to be included in the possible values of the return value. Furthermore, because c can be greater than 0 when the offset equals the length, the loop will not stop at this point, the offset will increase to infinity, and we will get the same result as before - a false alarm and all values between 0 and 14 as possible return values.

The reason for this behavior is that the join of convex polyhedra is not distributive. Therefore, when our translation turns one if into two, we lose information due to the interim join.

To conclude, although many false alarms exist, they are not a result of the interprocedural integer analysis algorithm, but rather due to a non-exact translation from C to IP.

Chapter 6

Related Work

This work is contributive in two main areas: interprocedural analysis and static detection of string errors.

6.1 Static Detection of String Errors

Many academic and commercial projects produce practical tools that detect string manipulation errors at runtime, e.g., [16, 1, 22, 7]. The main disadvantage of runtime checking is that its effectiveness strongly depends on the input tested, and it does not ensure against future bugs on other inputs. Our goal is to achieve a conservative static tool that detects all string errors and provides an assurance against all such errors.

Although the problem of string manipulation safety checking is to verify that accesses are within bounds [19, 3, 26], the domain of string programs requires that the analysis be capable of tracking the following features of the C programming language:

- 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); this length is dynamically changed based on the index of the first null character;
- simultaneously analyzing pointer and integer values, which is required in order to precisely handle pointer arithmetic and destructive updates.

Many academic projects produce unsound tools to statically detect string manipulation errors. In [20] an extension to LCLint is presented. Unsound lightweight techniques, heuristics, and in-code annotations are employed to check for buffer overflow vulnerabilities. Eau claire [5], a tool based on ESC-Java, checks for security holes in C programs by translating a subset of C to guarded commands. Its annotation language is similar in a sense to CSSV. In [29] 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 we target a tool that does not miss any error. Furthermore, none of them can track effects of pointer arithmetic, a widely used method for string manipulation. Sound algorithms for statically detecting string errors are presented in [12, 28]. However, they cannot handle all C, in particular multi-level pointers and structures. In [13] CSSV is presented which is a sound tool that handle all C and in a rather precise manner. However, CSSV analyze procedure calls by requiring the user to provide contracts. This can be a difficult task for large programs. This work is does not require contracts and handles procedure calls by a precise and efficient interprocedural analysis.

6.2 Interprocedural Pointer and Integer Analysis

The algorithm presented in this thesis is a variation of the "functional approach" to interprocedural static analysis, introduced in [27]. Our algorithm was also inspired by the functional approach and the coincidence theorem from [18]. The usage of 2-vocabulary to represent procedures summaries was inspired from shape analysis. This technique was used in [17].

In [8], interprocedural integer analysis using the polyhedra abstract domain was used for parallelizing FORTRAN programs. The algorithm presented here achieves improved accuracy over PIPS by using the polyhedra intersection operator as the abstract meet operator in return-sites. Furthermore, iCSSV handles a more complex problem than PIPS, because of the usage of pointers in C and iCSSV requirement to track both the indexes to strings and the string contents, i.e. the location of the null character.

Chapter 7

Conclusion

Buffer overflow is one of the most harmful sources of defects in C programs. Moreover, it makes software vulnerable to hacker attacks. We believe that iCSSV provides evidence that sound analysis can be automatically applied to statically verify the absence of all string errors in realistic applications.

References

[1] 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.

[2] R. Bagnara, E. Ricci, E. Zaffanella and P.M. Hill. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *Static Analysis Symp.*, 2002

[3] 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.

[4] CERT/CC. Vulnerability note vu♯317350. June 2004.

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

[6] 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.

[7] 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.

[8] B. Creusillet and F. Irigoin. Interprocedural analyses of Fortran programs. In *Parallel Computing* 24(3–4), pages 629–648, 1998.

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

[10] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Static Analysis Symp.*, 2001.

[11] N. Dor. *Automatic Verification of Program Cleanness*. PhD thesis, Univ. of Tel-Aviv, Israel, 2003.

[12] 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.

[13] 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.

[14] 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.

[15] M. Howard and D. C. LeBlanc. *"Writing Secure Code, Second Edition"*. Microsoft Press, 2002.

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

[17] B. Jeannet, A. Loginov, T. Reps and M. Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symp.*, 2004.

[18] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Comp. Construct.*, pages 125–140, 1992.

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

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

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

[22] 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.

[23] Microsoft Research. AST-toolkit. 2002.

[24] 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.

[25] C. Morgan. *Programming from Specifications*. Prentice-Hall, Engelwood N.J, 1990.

[26] 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.

[27] M. Sharir and A. Pnueli Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editores: *Program Flow Analysis*. Prentice Hall International, 1981.

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

[29] 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.

[30] D. A. Wheeler. *"Secure Programming for Linux and Unix HOWTO"*. www.dwheeler.com/secure-programs, 2003.

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

List of Figures

List of Tables