CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C

Nurit Dor[*]
Tel-Aviv University
nurr@post.tau.ac.il

Michael Rodeh
IBM Research Lab in Haifa
rodeh@il.ibm.com

Mooly Sagiv
Tel-Aviv University
msagiv@post.tau.ac.il

ABSTRACT

Erroneous string manipulations are a major source of software defects in C programs yielding vulnerabilities which are exploited by software viruses. We present **C** **S**tring **S**tatic **V**erifyer (CSSV), a tool that statically uncovers *all* string manipulation errors. Being a conservative tool, it reports all such errors 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. For this, procedures' *contracts* are allowed which are verified by the tool.

We implemented a CSSV prototype and used it to verify the absence of errors in real code from EADS Airbus. When applied to another commonly used string intensive application, CSSV uncovered real bugs with very few false alarms.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, Reliability, Validation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions, Pre- and post-conditions*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics, Program analysis*

General Terms

Algorithms, Reliability, Experimentation, Security, Languages, Verification

Keywords

Error detection, abstract interpretation, static analysis, buffer overflow, contracts

1. INTRODUCTION

String-manipulation errors are a common source of software defects and lead to many security vulnerabilities. CERT advisories report security holes resulting from *buffer overflow*, i.e., updates beyond the bounds of a buffer [29]. Furthermore, 60% of the UNIX failures reported in the 1995 FUZZ study [23] 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.

Our goal is to perform static analysis that detects *all* string runtime errors with just a few *false alarms*. A false alarm is a reported error that can never occur at runtime. This goal is ambitious. Existing methods either: (i) miss errors (e.g., LCLint [18], Eau Claire[2], and [29]); (ii) yield many false alarms (e.g., [18, 29]); or (iii) cannot handle com-

[*]Partially supported by a grant from the Ministry of Science, Israel and by the RTD project IST-1999-20527 "DAEDALUS" of the european FP5 programme.

plicated aspects of C, such as multilevel pointers and structures (e.g., [9, 28]). Moreover, the cost of static analysis is considered prohibitive when it comes to large programs.

This paper presents **C** **S**tring **S**tatic **V**erifyer (*CSSV* for short) — a tool that demonstrates that uncovering all string problems in C is achievable. CSSV is capable of analyzing realistic procedures and produces rather precise results. Being a conservative static-analysis tool it can never miss a runtime string error. It therefore guarantees the absence of *all* errors at the expense of sometimes generating false alarms.

For every procedure, CSSV allows 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, (such as the number of allocated bytes) that are defined by an *instrumented concrete semantics*.

1.1 Analysis of String Errors: CSSV

Fig. 1 shows how CSSV operates. Each procedure is analyzed separately. In the first phase, a source-to-source semantic-preserving transformation is applied to the analyzed procedure P. This transformation exposes the behavior of the procedures invoked by P by essentially inlining their contracts. The generated program yields a runtime error when a contract is violated. In addition, the inliner normalizes the C code to only include statements in a C subset called *CoreC* [30] which simplifies the task of implementing CSSV.

The second phase of CSSV analyzes pointer interactions. Conducting pointer analysis in a language like C is a nontrivial task. Moreover, it is difficult for programmers to define contracts regarding pointer behavior. Fortunately, several flow-insensitive algorithms have been shown to run on whole applications of considerable size, e.g., [4, 13]. Therefore, CSSV does not require contract information above pointers. Instead, CSSV applies a whole-program flow-insensitive pointer analysis to detect statically which pointers may point-to the same base address. 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 not reachable from visible variables of P cannot effect the postcondition of P. In many cases this allows subsequent analyses to perform strong updates when analyzing the procedure's body. We also compute certain *must-aliases* to improve the precision of the global flow-insensitive pointer analysis. Procedural points-to information was also used to improve the cost and precision of interprocedural analysis [21, 17, 6, 20].

In the third phase, the procedure's 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.

In the fourth phase, 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.

As opposed to alternative interprocedural program analy-

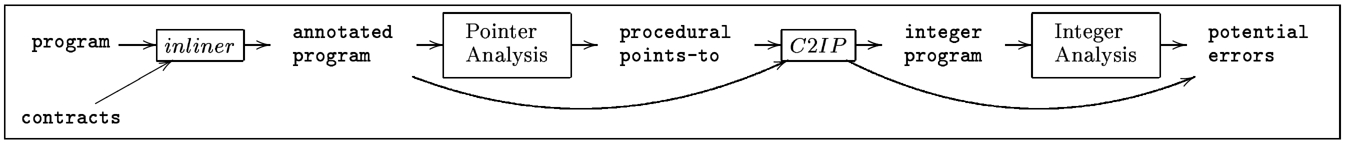

Figure 1: High-level structure of CSSV.

sis techniques, CSSV's approach has important advantages: (i) Each potentially recursive procedure can be analyzed separately, exactly once, i.e., the tool is applicable even if not all the source code is available (e.g., library functions). (ii) Contracts offer user control in a way similar to "design by contract" [25]. In particular, it enables CSSV to more effectively locate the actual source location in which the error occurs. (iii) Contracts can improve the precision of the analysis by providing information which can be hard to statically infer via an interprocedural analysis. (iv) By using the contracts to analyze procedure calls, CSSV applies a rather precise intraprocedural algorithm to reduce false alarms.

1.2 The Burden of Contracts

Contracts exert additional burden on the programmer. In the case of CSSV, this deficiency is minimized because the pre- and post-conditions need not describe the procedure's complete behavior. In particular, pointer information is automatically collected by CSSV, and therefore contracts usually omit information about how pointers are used. Moreover, unlike tools such as Eau Claire and LCLint, CSSV does not require annotations within the code itself such as loop invariants. Also, unlike these unsound approaches, since CSSV is sound, with any given contract, runtime errors cannot go undetected. Depending on the contracts, errors will be identified when analyzing the body of the procedure or at the procedure invocations. Of course, when a procedure's code is omitted such as in the case of library functions, CSSV assume its contract is correct and can not verify it.

Interprocedural modification side-effect analysis algorithms already exist (e.g., [27]). They can generate automatically the modify clause. Therefore, it is always possible to run CSSV with *vacuous contracts* including only the side-effect and a `true` pre- and post-condition.

This paper presents preliminary algorithms for automatically strengthening the pre- and post-conditions. The effectiveness of these algorithms is measured by comparing the number of false alarms obtained: (i) with the vacuous contracts, (ii) when using automatically derived contracts, and (iii) when using manually provided contracts.

The derivation procedure uses a forward sound integer-analysis algorithm called *ASPost* to compute an Approximation to the Strongest Postcondition of the integer program. Similarly, a backward sound integer-analysis algorithm called *AWPre* is used to compute an Approximation to the Weakest liberal Precondition [7]. The generated postcondition (precondition) is not necessarily the strongest because information is lost during the static integer analysis. Both ASPost and AWPre yield integer conditions. Therefore, the process can be repeated iteratively by running the derivation process given the generated integer conditions and further restricting the existing precondition (postcondition).

We also present a conservative method that uses the procedural points-to information to convert an integer expression for postcondition (precondition) into a C expression that can be used to strengthen the initial contract.

1.3 Main Results

The contributions of this paper are summarized as follows:

- 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 our previous algorithm, presented in [9], it handles the *full* spectrum of C language constructs, including dynamically allocated structures, multilevel arrays, multilevel pointers, function pointers, and casting. In addition, this algorithm is an order of magnitude better in its asymptotic time and space requirements.

- Preliminary program-analysis algorithms for strengthening pre- and post-conditions are presented. The algorithms reduce the burden on the programmer. They analyze the input procedure using existing (potentially vacuous) contracts and yield a new, more restrictive, contract for this procedure.
- We have implemented CSSV using the AST-Tooklit [22], CoreC, the Golf pointer analysis [4, 5], and the polyhedra integer analysis of [3] from [14]. We have applied the implementation to real-life programs. CSSV verified an intricate string library from EADS Airbus yielding only 6 false alarms. In the application fixwrites, part of web2c, CSSV uncovered 8 errors with 2 false alarms. Finally, we implemented the derivation algorithms and applied them to automatically generate pre- and post-conditions. The results show that in some cases it derived contracts equivalent to the manually specified ones.

1.4 Outline of the Rest of this Paper

The rest of the paper is organized as follows: Section 2 introduces CoreC, a contract language, a running example, and an instrumented concrete semantics. Section 3 describes CSSV. Section 4 describes the contract derivation algorithms. Section 5 describes the prototype implementation and the experimental results. Section 6 discusses related work, and Section 7 concludes the paper.

2. BACKGROUND

2.1 CoreC

CoreC is a subset of C with the following restrictions: (i) Control-flow statements are either `if`, `goto` , `break` or `continue`; (ii) expressions are side-effect free and cannot be nested; (iii) all assignments are statements; (iv) declarations do not have initializations; (v) address-of formal variables is not allowed. An algorithm for transforming C programs to CoreC is presented in [30]. Given a C program, it generates an equivalent CoreC program by adding new temporaries. CSSV is defined and implemented for CoreC. In the rest of this paper, CoreC is used instead of C.

2.2 Contracts

Contracts are used to describe expected inputs, side-effects, and expected output of functions. In this paper, we write contracts in the style of Larch [19]. Our implementation actually supports a more general executable language similar to [24], which can include loops. Contracts are specified in the `.h` file. Every prototype declaration of a function f has the form:

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

defining 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 arguments of f. We allow *attributes* of the form defined in Table 1 and displayed in Fig. 2. 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 specifying pointer values. In addition two shorthand expressions are allowed: (i) `string(arg)` — indicating that `arg` points to a null-terminated string, and (ii) `is_within_bounds(arg)` — indicating that `arg` points within the bounds of a buffer.

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 1: Attributes in the contract language.

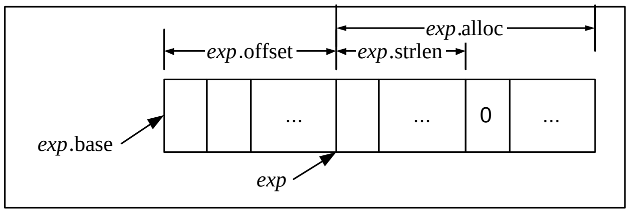

Figure 2: Graphical representation of contract-language attributes.

2.3 Running Example

The CoreC version of function `RTC_Si_SkipLine` from EADS Airbus (`SkipLine` for short) is shown in Fig. 3. `SkipLine` 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.

A contract for `SkipLine` is shown in Fig. 4. 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 at least zero. The function may modify the `*PtrEndText` pointer and the buffer pointed-to by `*PtrEndText`. The postcondition indicates that `*PtrEndText` points to a null-terminated string of length zero, and its value is advanced by `NbLine` bytes.

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 verify statically the absence of string errors in this function, without reporting any false alarm.

The toy `main` procedure, shown in Fig. 3, 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.

```
     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;
[1]     r = buf;
[2]     SkipLine(1,&r);
[3]     fgets(r,SIZE-1,stdin);
[4]     s = r + strlen(r);
[5]     SkipLine(1,&s); }
```

Figure 3: `SkipLine`, a string manipulation function from EADS Airbus with a toy main function.

There is a strong correlation between the contracts and the messages reported. However, errors do not go undetected. For example, omitting `NbLine >= 0` from the precondition yields an error message during the analysis of `SkipLine`. The message indicates that the postcondition

$$\texttt{*PtrEndText == } \lceil\texttt{*PtrEndText}\rceil_{pre} \texttt{ + NbLine}$$

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 can yield error messages on a procedure invocation. For example, requiring that `*PtrEndText` points-to a

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

Figure 4: A contract for `SkipLine`.

null-terminated string will cause an error message regarding the call to `SkipLine` at line [2] of `main`.

2.4 Instrumented Concrete Semantics

The C programming language does not define semantics for C programs. In the ANSI-C standard there is an informal notion of defined and undefined behavior. 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 semantics for C:

The address-of operation enables to change a variable's value without assigning to the variable. It also endures pointers to invisible variables.

Allocation (dynamic and static) routines of C provide an unformatted contiguous memory locations, while from the semantic point of view there is a "hierarchy" of objects where one object may contain objects of different types. Moreover, objects are type-less providing flexibility, where a location can be accessed according to different types. However, this causes difficultly in defining and checking the legitimacy of accesses.

"Big" L-values operations enables read and write of a number of primitives according to the type of an operand.

Pointer arithmetic is frequently used and has a defined result. However, checking its validity is impossible without additional instrumented information.

Cast operation is another well-defined feature but has many unclear outcomes. Moreover, a single location can be accesses (read or written) according to different types.

In this section, 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. In [8], the soundness of CSSV is proved with respects to this operational semantics. This semantics provides the foundation of CSSV's abstract interpretation.

Definition 2.1. *A* **concrete state at a procedure** P *is a tuple:* $state^\natural = (\mathcal{L}^\natural, \mathcal{BA}^\natural, aSize^\natural, loc^\natural, st^\natural, numBytes^\natural, base^\natural)$ *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* $\mathcal{L}^\natural$.
- $aSize^\natural : \mathcal{BA}^\natural \to \mathbb{N}$ *defines the allocation size in bytes of the memory region starting at a base address.*
- $loc^\natural : visvar_P \to BA^\natural$ *maps variables into their assigned global or stack locations (which is always a base address).*
- $st^\natural : \mathcal{L}^\natural \to val$ *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 accesses; primitive refers to the set of C primitive type values.
- $numBytes^\natural : \mathcal{L}^\natural \to \mathbb{N}$ *defines for each location the number of bytes of the value stored starting at the location.*
- $base^\natural : \mathcal{L}^\natural \to \mathcal{BA}^\natural$ *maps every location to its base address.*

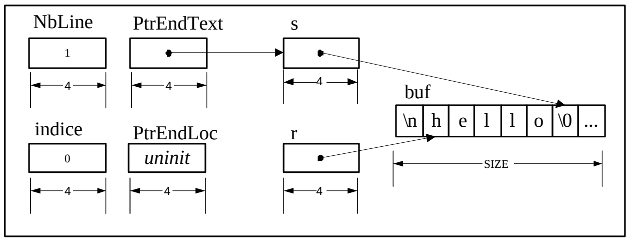

Figure 5: A concrete state arising at entry to `SkipLine` invoked by the second call from `main`. For clarity, the allocation size of `buf` is only shown symbolically.

A concrete state that arises at entry to `SkipLine` when invoked by the second call in `main` is shown in Fig. 5. Each box represents an allocated memory region starting at a base-address. We draw contiguous memory locations as boxes and display their allocation sizes underneath the boxes. Here, we assume that integers and pointers are four-byte long and a character is one-byte long. We draw a variable v above a box whose base address is $loc^\natural(v)$. The value inside each box shows the corresponding store content. Pointer values are drawn as edges.

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 $\mathcal{BA}^\natural$. The use of the $aSize^\natural$ and $base^\natural$ mappings allows the semantics to validate that pointer arithmetic and dereferences are within bounds. In order to handle destructive update to a variable via the address-of operation, $loc^\natural$ represents the address of variables, and $st^\natural$ maps locations into their values. For example, the pointer to `s` is described as a pointer to a location which is $loc^\natural(\texttt{s})$. Our concrete example contains, among others, the following interesting mappings:

$$\begin{array}{lcl}
st^\natural(loc^\natural(\texttt{PtrEndText})) & = & loc^\natural(\texttt{s}) \\
numBytes^\natural(loc^\natural(\texttt{PtrEndText})) & = & 4 \\
st^\natural(loc^\natural(\texttt{buf})+1) & = & \text{'h'} \\
numBytes^\natural(loc^\natural(\texttt{buf})+1) & = & 1
\end{array}$$

indicating that `PtrEndText` points-to the stack location of `s` which is a four-byte value, and that the second byte of `buf` contains the character 'h'.

The association of the number of bytes with locations enables us to handle cases where a location is accessed through different types. Specifically, writing a location as one type and later reading it as a different size type results in the *undefined* value.

Definition 2.2. *A concrete state* ($\mathcal{L}^\natural, \mathcal{BA}^\natural, aSize^\natural, loc^\natural, st^\natural, numBytes^\natural, base^\natural$) *is* **admissible** *if for every location* $l^\natural \in \mathcal{L}^\natural$ *such that*

$$st^\natural(l^\natural) \neq undefined$$

then

$$st^\natural(l^\natural + i) = undefined,\ 1 \leq i < numBytes^\natural(l^\natural) \wedge$$
$$\neg\exists l'^\natural < l^\natural : st^\natural(l'^\natural) \neq undefined \ \wedge \ l^\natural < l'^\natural + numBytes^\natural(l'^\natural)$$

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 to *undefined* all mapping that are overwritten.

L-values and *R-values* on C expressions can be defined by straightforward structural induction. In particular for a pointer variable p, we define the L-value and the R-value, denoted as l_p and r_p, respectively, as follows:

$$\begin{array}{rcl}
l_p(state^\natural) & \stackrel{\text{def}}{=} & loc^\natural(p) \\
r_p(state^\natural) & \stackrel{\text{def}}{=} & st^\natural(l_p(state^\natural)) \\
 & = & st^\natural(loc(p))
\end{array}$$

We define a function, $index^\natural$, to reason about the displacement of a location from its base. Formally,

$$\begin{array}{ll}
index^\natural : & \mathcal{L}^\natural \to \mathbb{N} \\
index^\natural(l^\natural) & \stackrel{\text{def}}{=} l^\natural - base^\natural(l^\natural)
\end{array}$$

With the additional information of $aSize^\natural$, $base^\natural$, the R-value of an attribute is easily defined. In particular the R-value of the attribute p.`offset` is $index^\natural(r_p(state^\natural))$.

Concrete states represent structures using sets of base addresses. Each field is associated with a unique base, $b^\natural$, and

$aSize^{\natural}(b^{\natural})$ is the size of the field. By abstracting this semantics, CSSV verifies that there are no accesses that use pointer arithmetic to cross field bounds.

3. CSSV

CSSV analyzes each procedure separately. We refer to the analyzed procedure as P. CSSV checks for three kinds of errors: (i) ANSI-C violations related to strings, such an access out of bounds. (ii) Violations of pre- and post-conditions of procedures as required by the provided contracts. When a procedure is invoked, the callee's precondition is checked. At the end of P, P's postcondition is checked. (iii) Our analysis checks certain *cleanness* conditions that correspond to good programming style. In particular, it validates that all accesses are before the null-termination byte, if it exists.

3.1 Technical Overview

Pointers and integers interact in a non-trivial way, especially in the C programming language. For example, it is non-trivial to check the safety of the expression

```
*PtrEndText = '\n'
```

in line [5] of `SkipLine`, i.e., that the pointer `*PtrEndText` is within bounds. CSSV infers the relationships between the offset of `*PtrEndText`, the allocation size of its base address, and the integer variables `indice` and `NbLine` needed to verify the safety of this destructive update. As we shall see, our algorithm statically verifies such inequalities by combining a pointer-analysis algorithm that detects pointers to the same base address, with an integer-analysis algorithm that detects *offset* relationships among pointers. The offset of a pointer is the index of the location it points to. Of course, in contrast to the concrete semantics, the abstract semantics summarizes many concrete locations by a single abstract location. It also maintains the potential points-to-relationships between these addresses.

CSSV applies a whole program flow-insensitive pointer analysis 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 `SkipLine` yields that `PtrEndText` may point to either `s` or `r`. Conservatively analyzing the function's body with the two calling contexts, requires treating updates to integer properties such as the offset of `*PtrEndText` as weak updates. 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 offset 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.

The procedural points-to information is used by C2IP to generate an integer program. Integer constraint variables summarize the semantic properties, (e.g., allocated size) of the represented locations. Finally, a conservative integer analysis determines potential values of the semantic properties and verifies the constraints upon them.

The rest of this section is organized as follows: Section 3.2 describes the procedure that inlines contracts in P. Section 3.3 formalizes the procedural points-to information for P. Section 3.4 describes the C2IP transformation applied to P. Section 3.5 sketches the integer-analysis algorithm.

3.2 Exposing Procedures' Behavior

The first step of CSSV takes as input the C program and the provided set of contracts, and generates a new C procedure $inline(P)$ by exposing the contracts of P and of the invoked procedures. Since $inline(P)$ contains `assert` statements that verify contracts, the behavior of $inline(P)$ differs from the behavior of P for inputs violating contracts. For other inputs $inline(P)$ and P behave the same.

Most of the C statements remain intact. Table 2 shows the scheme for translating the affected statements. We add the following syntactic extensions to C: (i) The construct `assume(`$\langle e\rangle$`)` that indicates that $\langle e\rangle$ holds after this statement, i.e., if $\langle e\rangle$ does not hold the execution is aborted without any message. It is used to reflect commitments of other procedures. (ii) Additional temporary variables named "$\langle e\rangle$" used to store the value of a subexpression $\lceil\langle e\rangle\rceil_{pre}$ at the procedure entry. (iii) the contract-language attributes which have a well-defined meaning in our instrumented concrete semantics.

Procedure entry is encountered before the first executable statement. In this case, the additional variables are initialized and P's precondition is verified. The designated variable `return_value` is set at every return statement. At every exit point (including `return`), P's postcondition is verified.

On a call to g we verify that g's precondition holds and assume that the postcondition holds. The original call to g is in the emitted code. This is essential for $inline(P)$ to behave the same as P.

3.3 Pointer Analysis

The second step of CSSV computes an abstraction of all potential pointer relationships between locations in concrete states that may occur during P's execution. However, only locations that can be accessed during the execution of P are of interest. Therefore, we define the notion of *reachable* locations.

DEFINITION 3.1. *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* $l^\natural$*).*

We infer the pointer relationships among reachable locations by computing a procedural pointer information that aims at representing the single location a formal points-to at the procedure entry. This section describes the abstract state representing pointer relationships and an algorithm to compute this state.

3.3.1 Procedural Points-to State

We formalize an abstract state that regards pointer relationships among reachable locations of P as follows:

DEFINITION 3.2. *A* **procedural abstract points-to state of** P *(PPT) is a quadruple* $state_P = (\mathcal{BA}_P, loc_P, pt_P, sm_P)$ *where:*

- $\mathcal{BA}_P$ *is a set of abstract locations that represent all reachable concrete base addresses.*
- $loc_P\colon visvar_P \to 2^{\mathcal{BA}_P}$ *maps variables into set of abstract locations representing the variable's global or stack locations.*
- $pt_P\colon \mathcal{BA}_P \to 2^{\mathcal{BA}_P}$ *abstract the possible pointers. A concrete pointer is represented by a* pt_P *relationship between the abstract locations representing the base addresses of the pointer's source and target location.*
- $sm_P\colon \mathcal{BA}_P \to \{1,\infty\}$ *is an abstract count on the number of concrete base addresses represented by an abstract location, i.e.,* $sm(ba) = \infty$ *when* ba *may represent more than one base address in a given concrete store, and* 1 *when it is guaranteed to represent at most one base address. An abstract location having* $sm = \infty$ *is a* **summary abstract location**. *Summary abstract locations can be used to represent unbounded sets of base addresses.*

We say that a PPT $(\mathcal{BA}_P, loc_P, pt_P, sm_P)$ is a *sound approximation of a concrete state* $(\mathcal{L}^\natural, \mathcal{BA}^\natural, aSize^\natural, loc^\natural, st^\natural, numBytes^\natural, base^\natural)$ in a procedure P if there exists a function $\alpha\colon \mathcal{BA}^\natural \to \mathcal{BA}_P$ satisfying the following requirements:

Base For all reachable $b^\natural \in \mathcal{BA}^\natural$: $\alpha(b^\natural) \in \mathcal{BA}_P$.

Stack For all $v \in visvar_p$: $\alpha(loc^\natural(v)) \in loc(v)$.

Pointer For all $l_1{}^\natural, l_2{}^\natural \in \mathcal{L}^\natural$ s.t., $l_1{}^\natural$ and $l_2{}^\natural$ are reachable, and satisfying $st^\natural(l_1{}^\natural) = l_2{}^\natural$: $\alpha(base^\natural(l_2{}^\natural)) \in pt(\alpha(base^\natural(l_1{}^\natural)))$.

Event	Emitted Code
entry of $P(f_1, f_2, \ldots, f_n)$	"$\langle e_i \rangle$" = $\langle e_i \rangle$; for every $\lceil \langle e_i \rangle \rceil_{pre}$ in $post[P]$ `assume`$(pre[P](f_1, f_2, \ldots, f_n))$;
`return` $\langle e \rangle$	`return_value`$_P = \langle e \rangle$;
`exit` P	`assert`$(post[P](f_1, f_2, \ldots, f_n))$;
$\langle e \rangle = g(a_1, a_2, \ldots, a_m)$	{ "$\langle e_i \rangle$" = $\langle e_i \rangle$; for every $\lceil \langle e_i \rangle \rceil_{pre}$ in $post[g]$ `assert`$(pre[g](a_1, a_2, \ldots, a_m))$; `return_value`$_g = g(a_1, a_2, \ldots, a_m)$; `assume`$(post[g](a_1, a_2, \ldots, a_m))$; $\langle e \rangle$ = `return_value`$_g$; }

Table 2: The emitted C code for effected statements. The notation $pre[x](e_1, e_2, \ldots, e_m)$ stands for the precondition of procedure x where formal f_i is replaced with the expression e_i. The expression $post[x]$ is obtained similar to $pre[x]$, however each of the $\lceil \langle e_i \rangle \rceil_{pre}$ expression is replaced with the variable "$\langle e_i \rangle$". `return_value`$_x$ is a designated variable representing the return value in the postcondition of procedure x.

Summary For all $b \in \mathcal{BA}_p$, s.t., $sm_P(b) = 1$, and $b_1^\natural, b_2^\natural \in \mathcal{BA}^\natural$ having $\alpha(b_1^\natural) = \alpha(b_2^\natural) = b$: $b_1^\natural = b_2^\natural$.

DEFINITION 3.3. *A $state_P$ is a* **sound approximation of** P *if it is a sound approximation of all the concrete states that may arise during the execution of P.*

L-values and R-values are generalized to return sets of abstract locations. In particular for a visible pointer variable q:

$$\begin{aligned} l_q(state) &\stackrel{\text{def}}{=} loc_P(q) \\ r_q(state) &\stackrel{\text{def}}{=} \textstyle\bigcup_{l \in l_q(state)} pt_P(l) \\ &= \textstyle\bigcup_{l \in loc_P(q)} pt_P(l) \end{aligned}$$

3.3.2 Constructing Procedural Information

CSSV computes a sound approximation $state_p$ in two stages: First, a whole-program analysis is applied to compute a global abstract points-to state of the whole program $Gstate = (\mathcal{BA}, loc, pt, sm)$ where:

- $\mathcal{BA}$ includes all abstract locations.
- loc: $var \to 2^{\mathcal{BA}}$.
- pt: $\mathcal{BA} \to 2^{\mathcal{BA}}$.
- sm: $\mathcal{BA} \to \{1, \infty\}$.

This global state is guaranteed to be a sound approximation of all procedures. Second, this global state is used to construct a sound approximation of P. Many possible solutions

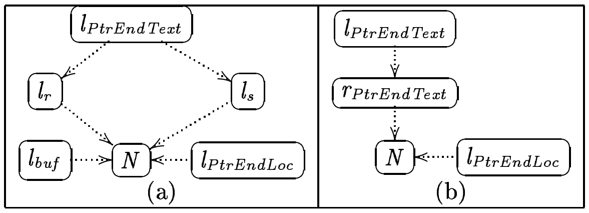

Figure 6: The whole-program points-to information for the running example (a), and the PPT for `SkipLine` (b).

exist. We decided to bias towards precise representation of formal parameters, with the intention to conduct strong updates on their properties in many cases.

Fig. 6 demonstrates the process. When possible we denote abstract locations as either as L-values (e.g., l_s) or as the R-value (e.g. r_{buf}) of some pointer. We can see that a new abstract location $r_{PtrEndText}$ represents the (unique) concrete location which holds the value of `*PtrEndText`.

Given a global abstract pointer state of the whole program $Gstate = (\mathcal{BA}, loc, pt, sm)$, let us construct, a PPT for P $state_P = (\mathcal{BA}_P, loc_P, pt_P, sm_P)$. The mapping loc_P is computed by projecting loc to the visible variables of P. Similarly, $\mathcal{BA}_P$ and sm_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 same state as the global points-to information shown in Fig. 6(a) without the l_{buf} abstract location. We aim at a potentially more precise

```
Boolean parameterizable( PPT $state_P$, formal $f$ )
{
  let $state_P = (\mathcal{BA}_P, loc_P, pt_P, sm_P)$
  let $l_f = loc_P(f)$ // the L-value of $f$
  if $sm_P(l_f) = \infty$ return false
  let $\{l_1, l_2, \ldots, l_m\} = pt(l_f)$ // the R-values of $f$
  for $i = 1$ to $m$
      if $sm_P(r_i) = \infty$ return false
      remove from $pt$ edges from $l_f$ to $l_j$ where $j \neq i$,
      and let $pt'$ be the resultant points-to map.
      if exists a reachable node $r_j, j \neq i$ in $pt'$ then
        return false
  // at most one of the concrete locations pointed-to by $f$
  // is reachable in a concrete state represented by $state_P$
  return true;
}
```

Figure 7: Algorithm to conservatively check that at most one concrete location is represented by the set pointed-to by a formal parameter f.

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 creating a new summary node is presented in Fig. 7. This algorithm checks that for every concrete store at most one concrete location is represented by r_f (the set of abstract locations pointed-to by a formal parameter f). The correctness of the algorithm is established in [8].

When permitted, the merge is done by (i) replacing $l_1, l_2, \ldots, l_m$ by a single non-summary abstract location r_f. (ii) setting $pt(r_f)$ to $\bigcup_{i=1}^{m} pt(l_i)$. This may improve the precision of destructive updates through f, but may decrease the precision of other updates.

3.4 C2IP

The C2IP transformation takes the *inline*(P) procedure with its PPT as input, and produces an integer program (IP for short) as output. The generated IP tracks the string and integer manipulations of P and of the invoked procedures. The IP is nondeterministic, reflecting the fact that not all values are known. The symbol *unknown* stands for an undetermined value. In particular, we use the following expressions:

`x :=` *unknown*;	Assigns any value to. `x`
`if` (*unknown*)	Either the true or the false branch can be taken.

The semantics of the `assume` construct in the integer program is to restrict the behavior of nondeterministic programs. Finally, for clarity, we use mathematical constructs in the IP.

The IP includes *constraint variables* used to denote interesting semantic properties such as offsets. C2IP 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 and for verifying contracts. In addition, C2IP can validate pointer assertions if specified in the contracts. Due to the flow insensitivity of our pointer analysis, this capability is rather weak in terms of precision. When a precondition may not hold, an error message is reported.

3.4.1 Constraint Variables

For every abstract location, l, C2IP generates the following *constraint variables*:

- $l.val$ 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.
- $l.aSize$, $l.is\_nullt$ and $l.len$ 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.

3.4.2 Translating Statements

Transforming C expressions involves querying the PPT to obtain the abstract locations a pointer *may* point-to. For simplicity, in this subsection we assume that every pointer may only point to a single non-summary abstract location.

C Exp.	Generated IP Condition
`*p`	$l_p.offset \geq 0 \wedge$ $((r_p.is\_nullt \wedge l_p.offset \leq r_p.len) \vee$ $(\neg r_p.is\_nullt \wedge l_p.offset < r_p.aSize))$
`p + i`	$l_p.offset + l_i.val \geq 0 \wedge$ $l_p.offset + l_i.val \leq r_p.aSize$

Table 3: Asserted IP conditions for C expressions.

Thus, l_p (representing the global or stack location of p), and r_p (representing the location pointed-to by p) are both singletons for every pointer p. In Section 3.4.2.3, the handling of arbitrary PPT is discussed.

3.4.2.1 Safety Checks.

For every basic C expression, there is a condition that verifies the validity of the expression. Table 3 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. CSSV follows [15, pp.205] and checks that the result of pointer arithmetic is either before or at the first location beyond the upper bound.

3.4.2.2 Update Statements.

C2IP generates statements to reflect semantic changes regarding the properties tracked. Table 4 displays the generated statements from transforming CoreC statements and conditional expressions involving pointers to buffers, which is the interesting part of C2IP.

On allocation, the resultant pointer always points to the 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 are before the null-termination byte (if it exists). Therefore, we can safely assume that when assigning a null-termination byte it is the first one. (ii) In the assignment of a non-zero character, it is checked if the null-termination byte was overwritten.

The generated IP does not contain function calls. Because C2IP transforms the *inline*(P) procedure, the pre- and post-conditions of an invoked procedure g are transformed. However, the call to a procedure needs to be analyzed conservatively. C2IP converts the call to g with the modify clause of g and substitutes actual for formal parameters. The modify clause is interpreted as assignments of *unknown* to the constraint variables of the abstract locations that represent potentially modified objects.

To increase precision, certain program conditions are interpreted. The second part of Table 4 shows the interpreted conditions. When checking for null-termination, C2IP replaces the condition with a condition over constraint variables that track length and the existence of a null-character. Pointer comparisons are replaced by expressions that compute the expression over the appropriate `offset` field.

For ease of use, the contract language allows specifying attributes on pointers instead of on base addresses. For example, `p.alloc` represents the allocation size starting at the location pointed to by p. The last part of Table 4 lists the transformation of contract's attributes to constraint variables by referring to the abstract locations pointed to by the specified pointer.

3.4.2.3 Other C Constructs.

The points-to graph contains *may* information, representing the fact that a pointer may or may not point to a specific location. Furthermore, all pointers to a summary abstract location are may pointers. To reflect the fact that a base address represented by l may or may not be modified, C2IP generates every update statement (shown in Table 4) as a nondeterministic assignment, under an `if` (*unknown*) statement. On the other hand, to be conservative, the analysis must take into account all possible values of a pointer, and verify expressions on all possible pointer values. This is done for all the generated `assert` statements and program conditions.

To handle casting and unions, C2IP generates for an assignment to one type of constraint variable assignments of *unknown* to the other constraint variables. For example, an assignment of an integer to a concrete location represented by abstract location l yields an assignment to $l.val$. In addition, C2IP generates the assignment $l.offset := unknown$. In particular, a cast to and from pointer type is conservatively handled by an assignment to *unknown*.

The pointer analysis determines which functions may be invoked at a call statement via a function pointer. Then, CSSV generates a non-deterministic statement that selects an arbitrary function call.

It is difficult to write general contracts for the format functions, such as `sprintf()` and `printf()`. Therefore, for the format functions, C2IP generates automatically pre- and post-condition according to the exact calling context. CSSV warns in cases where the format parameter is not a constant.

3.4.2.4 *The Complexity of C2IP.*

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. This is an order-of-magnitude improvement over the transformation in [9], which generates $O(V^2)$ variables and $O(S * V^2)$ statements.

3.5 Integer Analysis

The final step of CSSV analyzes the IP and reports potential assert violations. 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 to infer involve three

C Construct	IP Statements
`p = Alloc(i);`	$l_p.offset := 0;$ $r_p.aSize := l_i.val;$ $r_p.is\_nullt := false;$
`p = q + i;`	$l_p.offset := l_q.offset + l_i.val;$

`*p = c;`	if $c = 0$ then {
	  $r_p.len := l_p.offset;$
	  $r_p.is\_nullt := true;$ }
	else
	 if $r_p.is\_nullt \wedge l_p.offset = r_p.len$ then
	  $l_p.is\_nullt := unknown;$

`c = *p;`	if $r_p.is\_nullt \wedge l_p.offset = r_p.len$ then
	  $l_c.val := 0;$
	else $l_c.val := unknown;$

$g(a_1, a_2, \ldots, a_m);$	$mod[g](a_1, a_2, \ldots, a_m);$
`*p == 0`	$r_p.is\_nullt \wedge r_p.len = l_p.offset$
`p > q`	$l_p.offset > l_q.offset$
p.alloc	$r_p.aSize - l_p.offset$
p.offset	$l_p.offset$
p.is_nullt	$r_p.is\_nullt$
p.strlen	$r_p.len - l_p.offset$

Table 4: The generated transformation for C statements, conditional expressions and contracts' 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`.

and more properties, e.g., the postcondition of `SkipLine` regarding the new offset of `*PtrEndText`. Therefore, because our goal is as few as possible false messages, we apply the linear-relation-analysis algorithm [3, 12] 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 [3, 12, 9] for information about integer analysis.

3.5.1 *Assert checking*

During integer analysis, each `assert` statement is verified. This is done by checking if the asserted integer expression is implied by the linear inequalities that hold at the corresponding control-flow node. If it is not implied then

$$\begin{aligned} r_{buf}.aSize &= SIZE \\ r_{buf}.len &\geq 1 \\ r_{buf}.aSize &\geq r_{buf}.len + 1 \\ l_s.offset &= r_{buf}.len \end{aligned}$$

(a)

[5] SkipLine(1,&s);
require($r_{buf}.aSize - l_s.offset > 1$)

`error: the require may be violated when:`

$$r_{buf}.aSize = r_{buf}.len + 1$$

(b)

Figure 8: A report on the error in line [5] of `main`. The derived inequalities before execution of line [5] of `main` (a), and a counter example (b).

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. 8 demonstrates how the static integer-analysis algorithm identifies the error in the call to `SkipLine` in line [5] of `main`. The algorithm yields that the inequalities shown in Fig. 8 (a) hold before the execution of line [5], and that when the equality shown in Fig. 8 (b) holds a violation of `SkipLine`'s precondition occurs.

4. DERIVING CONTRACTS

This section presents integer-analysis algorithms to strengthen pre- and post-conditions. The following process is applied to a procedure P:

1. Compute side-effect information for P.

2. Run the inliner and C2IP with vacuous `true` pre- and post-condition which produces an integer program IP_0.

3. Run ASPost, a forward integer analysis of [3] on IP_0 which computes a safe approximation of the strongest postcondition. Obtain a new IP program IP_1 by strengthening the postcondition with the set of linear inequalities generated by the integer analysis at the procedure exit.

4. Run AWPre, a backward integer analysis on IP_1 which computes an approximation to the weakest liberal precondition. Obtain a new IP program IP_2 by strengthening the precondition with a set of linear inequalities generated by the analysis at the procedure entry.

5. *Writeback* — by using the PPT, convert the pre- and the post-conditions of IP_2 to C expressions over the formal parameters and global variables of P.

The derivation process can also start with manually given contracts. For applications with acyclic call graphs, the above process can be automatically applied in a bottom-up fashion, starting with the leaf procedures.

4.1 Integer Analysis

The ASPost algorithm is essentially the algorithm of Section 3.5 without reporting false alarms. It computes linear inequalities that hold at the exit point. Local variables are eliminated. The resulting inequalities are added to the input postcondition.

To improve the effectiveness of the derivation, the inliner phase is allowed to add designated variables to record values of properties that may be modified by P. For every potentially modified integer property expressed as a C expression $\langle e \rangle$, the *inline*(P) procedure includes a new variable "$\langle e \rangle$" with an additional C statement

$$\texttt{assume}(\text{"}\langle e \rangle\text{"} == \langle e \rangle);$$

During the writeback process, this variable is replaced by an appropriate $\lceil \langle e \rangle \rceil_{pre}$ expression in the postcondition. In this example, since `*PtrEndText` may be modified, variables are used to record all its properties. In particular, a variable "$r_{PtrEndText}.offset$" records the value of the expression $r_{PtrEndText}.offset$ at the entry.

The linear relationships obtained by ASPost when applied to `SkipLine` in the running example with a `true` precondition are:

$N.is\_nullt = \texttt{true}$

$N.len = r_{PtrEndText}.offset$

$$r_{PtrEndText}.offset \geq \text{“}r_{PtrEndText}.offset\text{”} + l_{NbLine}.val \quad (1)$$

The existence of a null-termination byte and the new length of the base address points to-by `*PtrEndText` is computed by ASPost precisely. ASPost finds a relationship between the old and new offsets of `*PtrEndText`. However, this relationship is weaker than the manually provided one on which the inequality is an equality. Both ASPost and AWPre may lose information due to joins of control-flow paths and due to the widening operation.

AWPre is similar to the forward algorithm in the sense that it uses the same abstract domain and abstract operations. The main difference is the treatment of assignments, which are handled by substitutions.

4.2 Write Back

The pre- and post-conditions generated by AWPre and ASPost are converted into C expressions over the formal parameters and global variables of P. These expressions are added to the input contracts using logical-and operator.

4.2.1 Obtaining Postconditions

Recall that the integer analysis computes properties of abstract locations. Each such abstract location corresponds to a set of L-value expressions over global and formal variables of P. Consider an abstract location l and assume, for simplicity, that there is a unique expression, say e, whose L-value is l. In this case, every constraint variable in the inequalities that occur in the exit are replaced by substituting e for a. Each occurrence of a designated formal parameter “$\langle e \rangle$” is replaced by $\lceil \langle e \rangle \rceil_{pre}$. Finally, the semantic properties are converted to the contract-language attributes.

For the equations in (1), the writeback algorithm yields:

**PtrEndText.is_nullt &&

**PtrEndText.strlen = 0 &&

*PtrEndText.offset >= $\lceil$`*PtrEndText.offset`$\rceil_{pre}$ + NbLine

When an abstract location corresponds to a set of L-value expressions, we generate a weaker postcondition using logical-or operator. An alternative would be to ignore some of these expressions, which may lead to false alarms when the procedure is analyzed by CSSV.

4.2.2 Obtaining Preconditions

Generating C expressions for preconditions from the entry inequalities is similar to the process of generating postcondition. The main difference is that we use logical-and instead of logical-or when multiple expressions correspond to the same abstract location.

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 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. Partial must information on pointer aliases is extracted from these edges. Both the integer analysis and the automatic derivation of pre- and post-condition were implemented using the Poly library.

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 5 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 for 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.

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.

There are six false messages on Airbus's code. 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.

The **Deriving** columns provides information about the effectiveness of the AWPre and ASPost algorithms. It is not trivial to measure the result in terms of precision. A new contract for a function P can change the result of the analysis of P itself and of procedures invoking P. We provide a simple measurement that is independent of the calling context. We run ASPost to generate a postcondition, AWPre to generate a precondition, and then run CSSV. Columns **CPU** and **Space** display the running time and total allocated space of both ASPost and AWPre. Column **Vacuous** displays the number of false-alarm messages reported by CSSV when a vacuous contract for the analyzed procedure is provided. Column **Auto** displays the number of false alarms reported by CSSV when using the automatically derived contracts. On average, the manually provided contracts reduce the number of false alarms by 93% as compared to the vacuous contracts' false alarms, while the automatic derivation algorithm reduces the number of message by 25%. The derived preconditions are in many cases weaker than the manually provided ones. Our initial study indicates that this happens when the integer analysis joins two different procedure behaviors. One potential remedy to this imprecision is by using sets of linear inequalities that allow to precisely represent logical-or.

6. RELATED WORK

6.1 Static Detection of String Errors

Although the problem of string-manipulation safety checking is to verify that accesses are within bounds [16, 1, 26], the domain of string programs requires that the analysis be capable of tracking the following features of the C programming language: (i) handling standard C functions, such as `strcpy()` and `strlen()`, which perform an unbounded number of loop iterations; (ii) 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; and (iii) simultaneously analyzing pointer and integer values is required which precisely handle pointer arithmetic and destructive updates.

Many academic projects produce unsound tools to statically detect string-manipulation errors. In [18] an extension to LCLint is presented. It uses unsound lightweight

App.	Function	Source Code			CSSV				Msg		Deriving			
		LOC	**SLOC**	**Contract**	**IP Vars**	**IP Size**	**CPU** sec	**Space** MB	**False**	**Errors**	**CPU** sec	**Space** MB	**Vacuous**	**Auto**
EADS Airbus	RTC_Si_SkipLine	13	260	SBI	39	109	2.6	12	0	0	0.3	3	5	5
	RTC_Se_CopieEtFiltre	66	773	SB	127	812	206	347	6	0	95	433	24	24
	RTC_Si_FiltrerCarNonImp	19	114	S	13	151	0.3	2	0	0	0.2	2	4	4
	RTC_Si_Find	26	820	SI	108	476	2.7	24	0	0	1.4	54	4	1
	RTC_Si_StrNCat	8	299	SBI	54	182	0.9	6	0	0	0.2	3	2	0
	RTC_Si_CalculerStringTime	33	567	SBI	86	529	76	127	0	0	131	173	21	4
	RTC_Si_FormatMcduTo-Formatprinter	18	273	SB	58	323	6.9	28	0	0	6.8	27	9	9
	RTC_Si_StoreIntInBuffer	35	222	SBI	59	346	9.8	43	0	0	3.3	22	15	15
	RTC_Se_ComposerEntete	10	550	SI	77	352	3.4	23	0	0	1.3	12	2	0
fixwrites	remove_newline	12	260	S	35	203	0.1	2	0	0	0.61	1	1	0
	insert_long	14	367	SB	138	571	13	99	0	2	23.4	86	5	0
	join	15	701	SB	95	443	2.1	23	0	2	6.7	15	2	2
	whole	30	423	S	46	352	1.2	20	0	1	0.6	4	9	9
	skip_balanced	20	258	SB	29	215	0.3	5	2	0	0.6	3	6	6
	bare	26	333	S	41	319	0.6	12	0	3	0.4	9	11	11

Table 5: The experimental results.

techniques, heuristics, and in-code annotations are used to check for buffer overflow vulnerabilities. Eau claire [2], a tool based on ESC-Java, check for security holes in C programs by translating a subset of C to guarded commands. It's annotation language is similar in 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 CSSV does not miss any error. Furthermore, none of them can track effects of pointer arithmetic, a widely used method for string manipulation. Sound algorithm for statically detecting string errors are presented in [9, 28]. However, they can not handle all C, in particular multi-level pointers and structures. As far as we know, CSSV is the first sound tool to handle all C and in a rather precise manner.

6.2 The Automatic Derivation Process

The Houdini annotation-derivation tool [10] tries ESC/Java with different annotations. Such an approach is inadequate in our case because the number of potential annotations is unbounded. In contrast, we derive a contract by forward and backward analyses of the integer program [11].

7. CONCLUSIONS

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

Acknowledgments

We would like to thank Manuvir Das for providing and assisting us with AST-ToolKit and GOLF. Thanks to Bertrand Jeannet and Nicolas Halbwachs for providing us the polyhedra library and for their support. Thanks to Greta Yorsh for her assistance in the prototype implementation and for many technical insights. Thanks to Seth Hallem, Roman Manevich, Tom Reps, and Reinhard Willhelm for their helpful comments.

8. REFERENCES

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

[2] B. Chess. Improving computer security using extended

static checking. In *IEEE Symposium on Security and Privacy*, 2002.

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

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

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

[6] 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.

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

[8] N. Dor. *Statically Detecting All Buffer Overflows in C.* PhD thesis, Univ. of Tel-Aviv, Israel, 2003. In preparation.

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

[10] C. Flanagan, K. Rustan, and M. Leino. Houdini, an annotation assistant for Esc/java. In *Formal Methods for Increasing Software Productivity, volume 2021 of Lecture Notes in Computer Science*, 2001.

[11] N. Halbwachs. *Static Analysis of Linear Properties Invariantly Satisfied by the Numeric Variables of a program.* PhD thesis, Grenoble University, 1979.

[12] 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.

[13] 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.*, 2001.

[14] B. Jeannet. New polka library. Available at "http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html".

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

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

[17] W. Landi. *Interprocedural Aliasing in the Presence of Pointers.* PhD thesis, Dept. of Comp. Sci., Rutgers Univ., 1991.

[18] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, 2001.

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

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

[21] T.J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Symp. on Princ. of Prog. Lang.*, 1990.

[22] Microsoft. Ast toolkit. Available at "http://research.microsoft.com/sbt/asttoolkit/ast.asp", 2002.

[23] 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.

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

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

[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.*, 2000.

[27] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, 2001.

[28] A. Simon and A. King. Analyzing string buffers in c. In *International Conference on Algebraic Methodology and Software Technology*, 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] G. Yorsh. *CoreC: A Simplifier for C*, 2002. http://www.cs.tau.ac.il/∼ gretay/GFC.htm.