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

Nurit Dor, Michael Rodeh and Mooly Sagiv . PLDI'2003

(Lecture summary by Ofri Ziv and Ben Riva)

Motivation:

Out-of-bound access to an array is a very common error in C. Consider, for example, the following small piece of code:

```
/* from web2c [strpascal.c] */
void foo(char *s)
{
        while ( *s != ' ' )
                        s++;
        *s = 0;
}
```

Although this code looks simple, in some circumstances it might work differently than what the programmer had in mind. E.g., if *s* is not formatted or allocated properly, *s++* and **s=0* might point to unallocated memory or a memory that belongs to other objects.

Of course, this type of errors can cause many undesirable problems, such as:

1. Program crash (e.g., if the written memory is the return address of the function, and the written data is an invalid memory).
2. Program misbehaving (e.g., if the written memory belongs to another data structure that is overwritten).
3. Exploitation by a malicious user (e.g., hackers) that changes the program execution to some unintended execution flow.

Despite the potential devastating consequences, such errors are very common: FUZZ (http://pages.cs.wisc.edu/~bart/fuzz/) study shows that 18%-23% of tested UNIX programs hang or crash as a result of such error. CERT (Computer Emergency Response Team) claimed that up to 50% of attacks are due to buffer overflow.

Main Goal:

Detect ***all*** buffer overflows in C programs while getting only a ***minimal*** number of false alarms. In addition, provide the programmer with details about the identified error. All of this is done using static analysis techniques and by adding small code snippets to the original program.

In particular, CSSV detects the following types of string errors:

- Buffer overflow (update beyond bounds)
- Unsafe pointer arithmetic
- References beyond null termination

- Unsafe library calls

CSSV can handle even more complex uses of pointers such as pointer arithmetic, multi-level pointers, etc.

Last, to verify that CSSV works for real programs, evaluation is done on a public domain software *fixwrites* and on a string manipulation library from the Airbus system. CSSV is quite scalable since it works with granularity of procedures, making the static analysis steps more efficient.

Why is it difficult?

Consider the following code snippet:

```
/* from web2c [fixwrites.c] */
#define BUFSIZE 1024
char buf[BUFSIZE];
char insert_long(char *cp)
{
        char    temp[BUFSIZE];
        …
        for (i = 0; &buf[i] < cp ; ++i)
                        temp[i] = buf[i];
        strcpy(&temp[i],"(long)");
        strcpy(&temp[i+6],cp);
        …
}
```

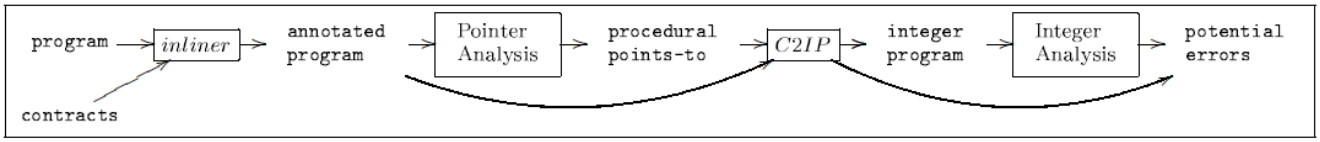
In the orange statement, overflow occurs if:
BUFSIZE - (cp - buf) < 8
In the red statement, overflow occurs if:
BUFSIZE – (cp – buf) - len(cp) < 8
These overflows are not easily detected by static analysis since the status of the pointer cp isn't always clear. Also, checking the absence of overflows requires checking many different constraints, e.g. see the constraints needed for the following simple code:

```
void safe_cat(char *dst,  int size, char *src )
{
        {string(src) ∧ string(dst)  ∧ (size > len(src)+len(dst))⇒ alloc(dst+len(dst)) > len(src))}
        if ( size > strlen(src) + strlen(dst) )
        {
                        {string(src) ∧ string(dst)  ∧  alloc(dst+len(dst)) > len(src)}
                        dst = dst + strlen(dst);
                        {string(src) ∧ alloc(dst) > len(src)}
```

```
            strcpy(dst, src);
        }
}
```

CSSV main steps:

- C to CoreC translation - CSSV translates the C code to CoreC code using a tool of Greta Yorsh (from TAU). CoreC is much simpler to analyze than pure C, which has many low level features. Although CoreC is only a subset of C, it is complete and the translation preserves all the semantics of C.
- Procedure contracts - The programmer specifies pre- and post-conditions of every procedure that should hold in all good executions. The specification is done by annotation in the code itself. These conditions are also translated to CoreC code that is inlined into the code from previous step. *All violations of these conditions are detected!*
- Points-to analysis - Static analysis of the CoreC code is used to identify points-to relations (including "widening" for loops). The analysis is flow-insensitive on the entire program, and a projection of it on each procedure is then computed.
- All pointer operations are abstracted per procedure resulting in constraints and inequalities that represent the relationships between the different pointers. These constraints and inequalities are fed into Polyhedra, in order to detect potential violations.



Contracts:

The programmer adds annotations, in the .h files of the C code, to every procedure he defines. That includes: (1) The pre-conditions, i.e., what are the assumptions made by the procedure; (2) The post-conditions, i.e. what assumptions the procedure guarantees to preserve; and (3) The side effects of the procedure (e.g., in case it manipulates external data structures).
Contracts as annotations allow a modular analysis that is more accurate and does not require availability of the entire code. Though, indeed, writing contracts require more work from the programmer.

Later on, when inlined into the CoreC code, the contract conditions are implemented using *assert()* and *assume()* calls. See the following procedure contracts examples:

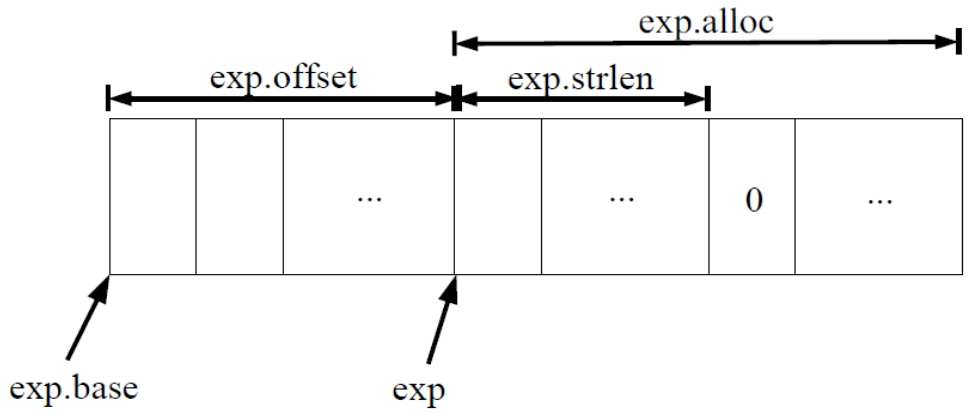
```
char* strcpy(char* dst, char *src)
        requires    ( string(src) ∧
                      alloc(dst) > len(src)
                    )

        mod         len(dst), is_nullt(dst)
        ensures     ( len(dst) = = pre@len(src) ∧
                      return = = pre@dst
                    )
```

```
void safe_cat(char* dst, int size, char* src)
        requires    ( string(src) ∧ string(dst)
                      alloc(dst) == size
                    )
        mod         dst
        ensures     ( len(dst) <= [len(src)]_pre +
                                  [len(dst)]_pre ∧
                      len(dst) >= [len(dst)]_pre
                    )
```

As you can see from these examples, each procedure has pre-conditions (“requires”), post-conditions (“ensures”) and potential side-effects (“mod”). The statements in each part can include any C expression that doesn't call other procedures and refer only to the procedure's or/and global variables.

In order to abstract the buffers state, several attributes of pointer abstraction are added to the language. Say *exp* points inside some buffer, then the following attributes are available:



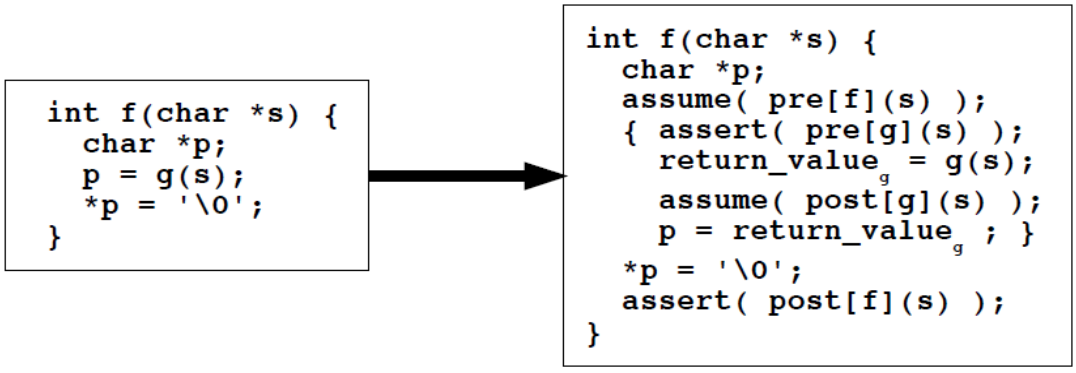
That is, *exp.base* is the base address of the allocated buffer, *exp.offset* is the offset of exp from *exp.base*, *exp.len* is the number of chars until a null character (if existed) and *exp.alloc* is the length of allocated buffer from *exp.offset*. Also, *exp.is_nullt* is a true if exp points to a null terminated string. Other shorthands are *string(arg)*, indicating *arg* points to a null terminated string, and *is_within_bounds(arg)*, indicating that *arg* points within the bounds of a buffer. See, for example, the next procedure and its contract:

```
   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); }
```

```
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 ;
```

As mentioned before, contracts are emitted to code that use *assert()/assume()* to check conditions on runtime. Simple expressions are checked directly by *assume()*. Expressions that depend on other expressions or states use temporary variables. For example, the above post-condition needs the value of **PtrEndText* from the pre-condition, so that value is stored in a temporary variable in the procedure prologue and can be used in the procedure epilogue. Similarly, any call to other procedure is checked too. See next diagram:



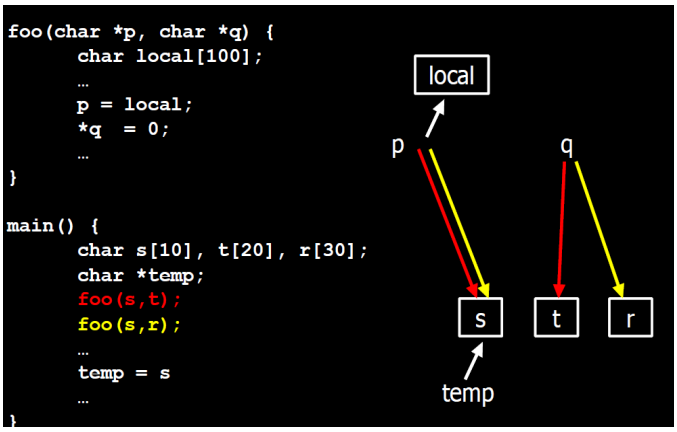
where *pre[f](s)/post[g](s)* denote the pre-/post-condition when calling *f/g* with *s*, respectively. (*return_value* represents the return value of the given procedure, and can be accessed from the contracts code.)

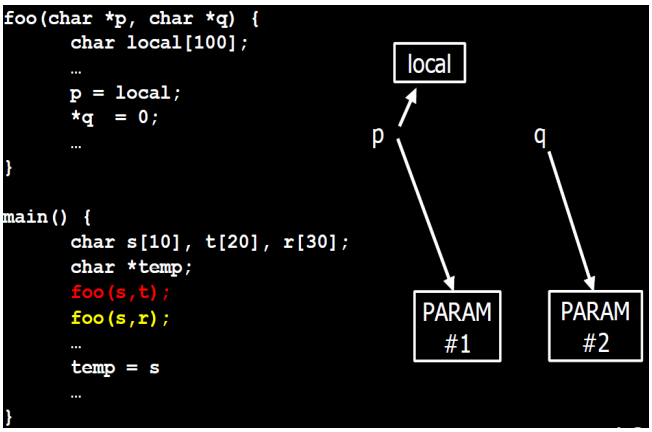
Last, we stress that bad contracts do not cause CSSV to miss buffer overflow but only might increase

the number of false alarms.

Pointer analysis:

First, flow-insensitive points-to analysis is done for the whole program and then projected to the procedure level. In the next example, the analysis on the whole program is on the left and its projection on *foo* is on the right.





Of course, not all numeric values are known during the analysis. Therefore, pointer states are abstracted and then checked on runtime and transformed into an integer program that is analyzed by Polyhedra. More specifically, denote by:

- *lp* the abstract location of pointer *p*
- *lp.val* the potential values stored in *lp*
- *lp.offset* the potential offset of the pointers represented by *lp*
- *lp.aSize* the allocation size
- *lp.is_nullt* the predicate if *lp* is a null-terminated string
- *lp.len* the length of the string.

All these are abstract values. Also note that null-terminated strings are handled more carefully to validate also the existence of null in the allocated buffer.

Checked conditions are generated using the above abstraction following the next transformers:

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	$l_p.offset + l_i.val \geq 0 \wedge$ $l_p.offset + l_i.val \leq r_p.aSize$

Then, integer relationships (i.e., set of offset inequalities) are constructed using the next rules:

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	$r_p.len - l_p.offset$

All these constraints and relationships are fed into Polyhedra in order to find potential integer values that violate them. For example, see the following table for the procedure *SkipLine()* we saw earlier:

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

(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)

The set of relations is in the top part of the table, and the result of Polyhedra finds a case in which a contract might be violated for the given set of relations, in the bottom part.

Evaluation:

CSSV was implemented using Microsoft AST Toolkit, Microsoft GOLF and the New Polka.

Results for string intensive procedures of the Web2C library are in the next table:

Proc	line	coreC line	time (sec)	space (Mb)	errors	FA
insert_long	14	64	2.0	13	2	0
fprintf_pascal_string	10	25	0.1	0.3	2	0
space_terminate	9	23	0.1	0.2	0	0
external_file_name	14	28	0.2	1.7	2	0
join	15	53	0.6	5.2	2	1
remove_newline	25	105	0.6	4.6	0	0
null_terminate	9	23	0.1	0.2	2	0

Results for Airbus code are in the following table:

Proc	line	coreC line	time (sec)	space (Mb)	errors	FA
FiltrerCarNonImp	19	34	1.6	0.5	0	0
SkipLine	12	42	0.8	1.9	0	0
StoreIntInBuffer	37	134	7.9	21	0	0

Note that CSSV found several real bugs in the Web2C library (see “errors” column). More interestingly, as can be seen from the “FA” columns, the number of false alarms is pretty small.

As for resources, space grows depending on the procedure's length, but since it can be done sequentially, the overall space needed is not too large. On the other hand, the computation itself is rather time consuming. However, note that this is done only once, when running CSSV, and not on runtime.