Applications of Program Analysis: CSSV and Shape Analysis

Ariel Jarovsky and Liron Schiff

Table of Contents

CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C, by Nurit Dor, Michael Rodeh, Mooly Sagiv.

Introduction

This work presents a static analysis scheme for C code. This pioneer work has influenced successive works that dealt with more scenarios, for example Microsoft's SAL.

The target of the scheme is to statically verify the absence of buffer overflows in C code with enormous string manipulations. This is a conservative tool that reports all such errors at the expense of sometimes generating false alarms.

CSSV takes a modular approach as it analyses each procedure separately. The disadvantage is that the programmer is required to supply a contract to each procedure.

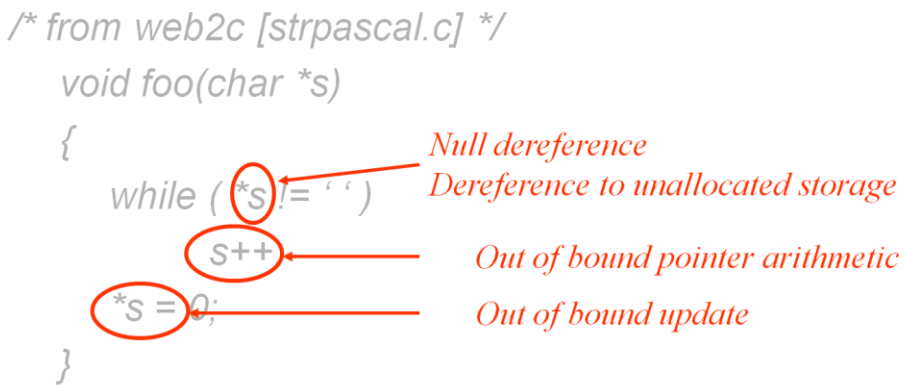
Possible overflows:

- Null dereference – trying to read a value pointed by Null pointer.

- Out of bound arithmetic – advancing a pointer beyond the buffer 's bounds[1]
- Out of bound update - writing to an address beyond the buffer 's bounds

The Airbus code itself is confidential so we will use open source examples.

Examples to buffer overrun:



Such overruns are common, FUZZ study has found that in 9 different UNIX systems, 18%-23% of hangs or crashes are caused by overruns. Another study, CERT, has shown direct connection between buffer overruns and software attack (50% of the attacks are based on buffer overflows).

We want an algorithm that is efficient, sound and fully supports C syntax. We also want it to produce a few as possible false alarms that are very expensive in critical applications such as aviation.

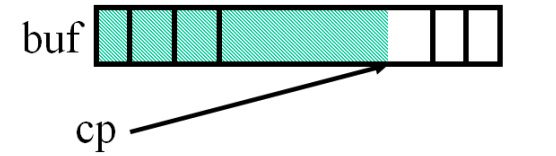
A complicated example:

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

char insert_long(char *cp)
{
      char    temp[BUFSIZ];
      ...
      for (i = 0; &buf[i] < cp ; ++i)
            temp[i] = buf[i];

      strcpy(&temp[i],"(long)");
      strcpy(&temp[i+6],cp);
      ...
```

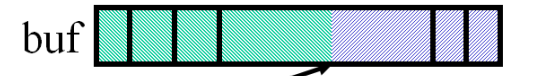
In this example there are two possible overflows:TODO

- In statement strcpy(&temp[i],"(long)") if 7 + offset (cp) ≥BUFSIZ

[1] ANSI C allows placing a pointer at maximum offset one beyond the bound of the buffer, for handling loops. In case the pointer exceeds by more than one a warning should be issued.



- In command strcpy(&temp[i+6],cp) if offset(cp)+7 +len(cp) ≥ BUFSIZ



Verifying absence of buffer overflow is non-trivial, the conditions may become complex. In the next example the conditions for the absence of buffer overflow are included:

```
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);
 }
}
```

These conditions are built in a bottom up manner. They use the following expressions:

- *string(src)* – indicates that *src* is null terminated.
- *alloc(dst)* – returns the size of buffer (or sub-buffer) pointed by *dst*.
- *strlen(src)* - returns the length of string (or sub-string) pointed by *src*.

The CSSV approach

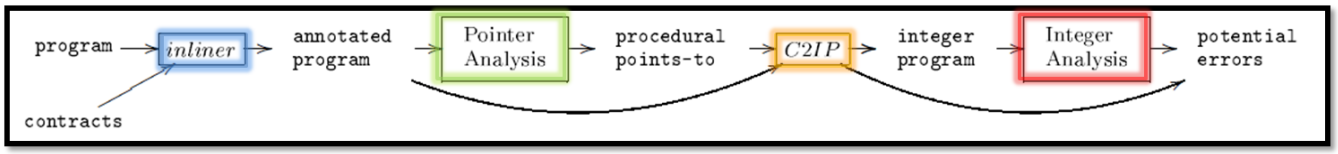
Challenges

Challenge	Selected Method	Details
Complex linear relationships	Use Polyhedra[CH78]	Linear inequalities
Pointer arithmetic	Pointer analysis	Point-To
Loops	Widening	Might cause false-alarms
Procedures	Procedure contracts	In/out conditions per procedure

Highlights

- Detects string violations
  - Buffer overflow (update beyond bounds)
  - Unsafe pointer arithmetic
  - References beyond null termination
  - Unsafe library calls
- Handles full C
  - Multi-level pointers, pointer arithmetic, structures, casting, …
- Applied to real programs
  - Public domain software
  - C code from Airbus

Technical Overiew



CSSV analyzes each procedure separately, it operates at the following 4 phases:

1. A source code transformer is applied (while preserving semantics) to the analyzed procedure P. this transformation essentially inlines the contracts specified by the programmer in the header file. In addition the inliner normalizes the C code to only include statements in a C subset called CoreC which simplifies the task of implementing CSSV.

2. The pointer interactions are analyzed. This is done by applying 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 point-to information for the analyzed procedure P. this algorithm benefits from the fact that memory locations not reachable from visible variables of P can not affect the postcondition of P. Certain must-aliases are computed to improve the precision of the global flow-insencitive pointer analysis.

3. The procedure's code and points to information are fed into the C2IP transformer. C2IP generates a procedure that manipulates integers. It transforms the CoreC code into another CoreC code which maintains pointer relations relevant for the *Inferred Concrete Semantics* of the program (for example, a pointer assignment x = y will be transformed in x.offset = y.offset and x.base = y.base). 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.

4. 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 post condition is not guaranteed to hold. For minimizing the number of false alarms, CSSV uses Polyhedra, an 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 the 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.

Procedure Calls – Contracts

Motivation

```
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
               )
```

One important aspect of CSSV is the use of contracts to handle each procedure by its own. This modular aspect enables the analysis to take place even when not all the code is available. It also allows the user to fine tune the analysis and its outcome. The user can decide to check additional properties and many errors are logically contract-related which helps to understand them.

One should have in mind that writing bad contracts cannot violate the soundness of the analysis[2], it can however generate more false alarms and makes the outcome less relevant.

Syntax

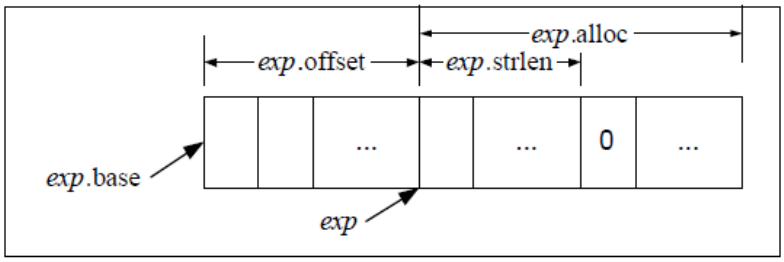
Contracts are specified in the **.h** file. Every prototype declaration of a function *f* has the form:

$$\langle type \rangle\ f\ (\cdots) \quad \textbf{requires} \quad \langle e \rangle$$
$$\textbf{modifies} \quad \langle e \rangle, \langle e \rangle, \ldots, \langle e \rangle$$
$$\textbf{ensures} \quad \langle e \rangle;$$

$\langle e \rangle$ is a C expression, without function calls, over global variables and the arguments of *f*. The following attributes are also available:

[2] We assume that the contracts are not used as replacement to unavailable code.

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*



The syntax $\lceil\langle e\rangle\rceil_{pre}$ denotes the value of $\langle e\rangle$ when *f* is invoked.

More shorthand expressions:

1. *string(arg)* – indicating that *arg* points to a null-terminated string.
2. *Is_within_bounds(arg)* – indicating that *arg* points within the bounds of a buffer.

Example

Function code:

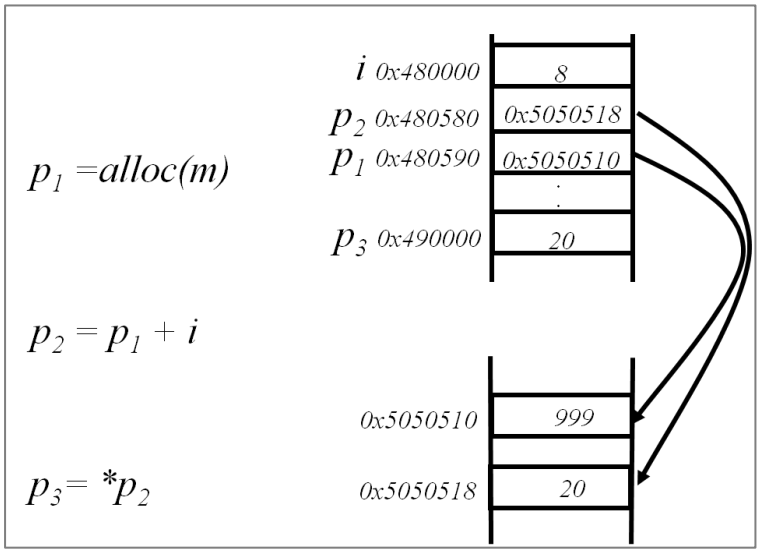
```
void SkipLine(int NbLine, char** PtrEndText)
{
   int indice;
   char* PtrEndLoc;
[1] indice=0;
[2] begin_loop:
[3] if (indice>=NbLine) goto end_loop;
[4] PtrEndLoc = *PtrEndText
[5] *PtrEndLoc = '\n';
[6] *PtrEndText = PtrEndLoc + 1;
[7] indice = indice + 1;
[8] goto begin_loop;
[9] end_loop:
[10]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

Function prototype with contract:

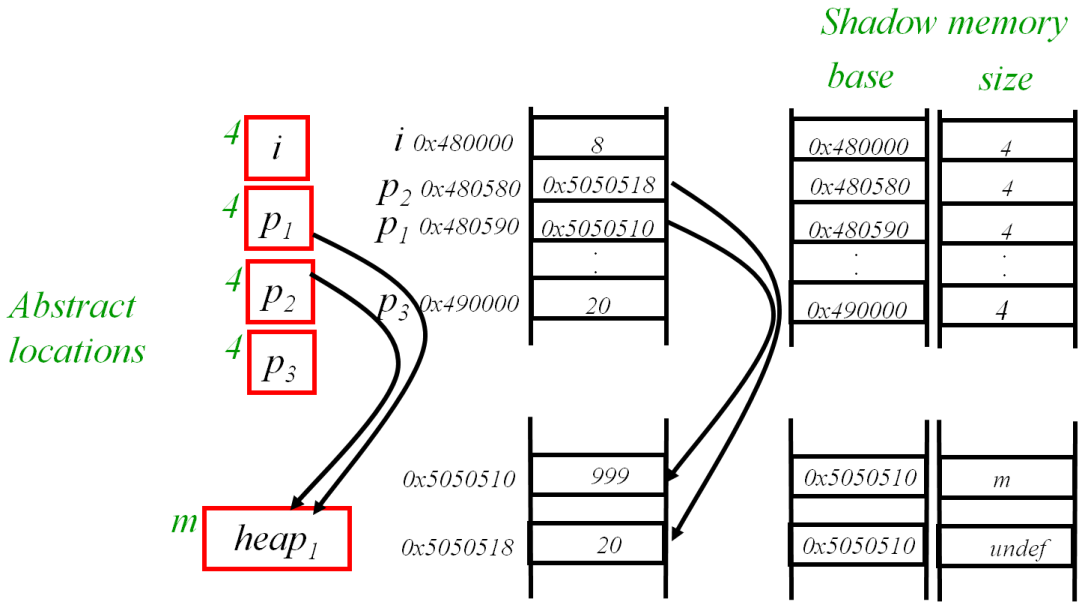
```
void SkipLine(int NbLine, char** PtrEndText)
  requires is_within_bounds(*PtrEndText) &&
      *PtrEndText.alloc > NbLine && NbLine >= 0
  modifies *PtrEndText.strlen,
      *PtrEndText.is_nullt, *PtrEndText
  ensures *PtrEndText.is_nullt &&
      *PtrEndText.strlen == 0 &&
      *PtrEndText == [*PtrEndText]pre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

Instrumented concrete semantics

Point-To expansion

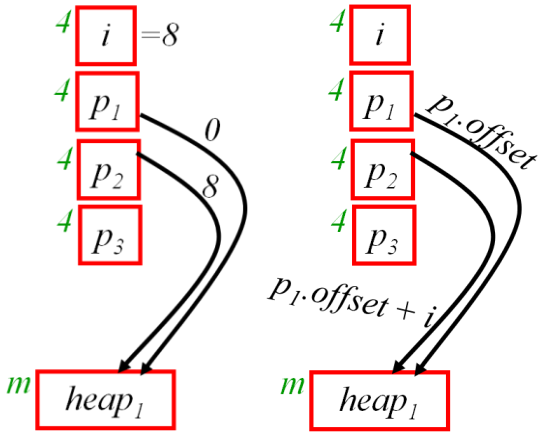


Simple point-to analysis, as we have seen in class, cannot be directly applied to C language in order to verify correctness. It can't express the continuality notion of allocated buffers. For example it can't ensure that pointer arithmetic is bounded inside a buffer.

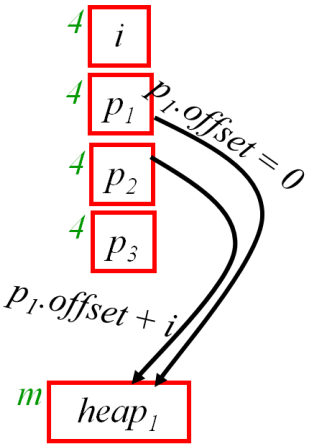


CSSV defines the C concrete semantics introducing the Shadow Memory that stores the base address and size for each location. CSSV extends Point-To abstraction by tracking also the size and base address for each variable, ignoring exact locations.

Pointer Validation



In order to validate pointer arithmetic, CSSV tracks offsets from origin for each pointer, and it tracks numeric values. When numeric values are unknowns CSSV tracks integer's relationships, for example, the statement $p_2 = p_1 + i$ when i is unknown will yield the relation $p_2.offset = p_1.offset + i$. These relations are used as steps in the integer program constructed by C2IP (in the 3rd phase).



We define the effect of the statement $p_1 = alloc(m)$ by the relations $p_2.offset = 0$ and $heap_1.size = m$.[3] we also support null termination tracking by adding the relation $heap_1.is\_nullt = false$.

More translations are defined in the paper.

We validate a pointer arithmetic statement such as $p_2 = p_1 + i$, by requiring an inequality of the form $*p_1.size \geq p_1.offset + I \geq 0$. We validate a pointer dereference statement such as $p_3 = *p_2$, by requiring an inequality of the form $*p_2.size \geq p_2.offset$. Such inequalities are

[3] $heap_1$ is actually $*p_1$ which appear as l_{p1} in the paper as the join over all possible abstract locations pointed by p_1. The location of p_1 itself appears as r_{p1} in the paper.

combined as asserts and checked in the last phase, against the polyhedral, yielding an error in case of possible violation.

Notice that we validate dereferences even though we validate all pointer arithmetic, this is because pointer arithmetic (according to ANSI C) is allowed to bypass buffer's upper bound by one.

Complete Example

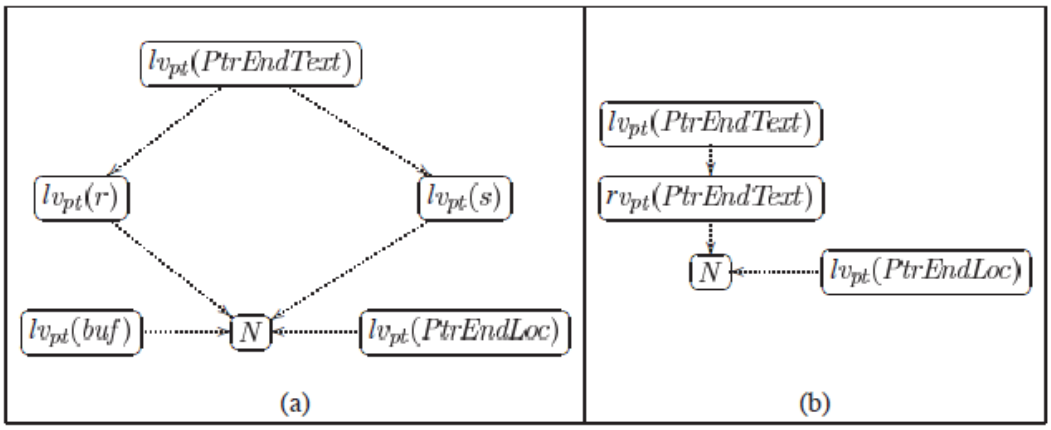
We will present the CSSV process on the SkipLine function:

The input function and contract:

```
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 ;
```

```
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); }
```

The points to result for the whole program and the induced part for SkipLine (where $lv_{pt}(x)$ and $rv_{pt}(x)$ are the possible locations of x and the possible location pointed by x corresponding):



The integer relations computed using the code, the points-to information and contract:

```
/* void SkipLine(int NbLine, char** PtrEndText) { */
/* requires is_within_bounds(*PtrEndText) && */
        assume( 0 ≤ rv_pt(PtrEndText).offset ≤ B.size );
        assume ( ¬B.string ∨ rv_pt(PtrEndText).offset ≤ B.len );
/*      *PtrEndText.alloc > NbLine && */
        assume( B.size − rv_pt(PtrEndText).offset > lv_pt(NbLine).val );
/*      NbLine >= 0 */
        assume( lv_pt(NbLine).val ≥ 0);
        "*PtrEndText.offset".Pre = lv_pt(PtrEndText).offset;
        "*PtrEndText.alloc".Pre = rv_pt(PtrEndText).size − lv_pt(PtrEndText).offset;
/* int indice; */
        lv_pt(indice).size := 4;
⋮
/* [1] indice=0; */
        if (lv_pt(indice).size = 4 ∨ lv_pt(indice).val = uninit) lv_pt(indice).val := 0;
        else lv_pt(indice).val := ⊤
⋮
/* [3] if (indice>=NbLine) goto end_loop; */
        if (lv_pt(indice).val ≥ lv_pt(NbLine).val) goto end_loop;
        else skip;
⋮
/* [5] *PtrEndLoc = '\n'; */
        assert (0 ≤ lv_pt(PtrEndLoc).offset < B.size ∧
          (¬B.string ∨ lv_pt(PtrEndLoc).offset ≤ B.len)
        if ( B.len = lv_pt(PtrEndLoc).offset )
          B.string = false;
        else skip;
/* [6] *PtrEndText = PtrEndLoc + 1; */
        rv_pt(PtrEndText).offset := lv_pt(PtrEndLoc).offset + 1;
⋮
/* [11] *PtrEndLoc = '\0';*/
        assert (0 ≤ lv_pt(PtrEndLoc).offset < B.size ∧
          (¬B.string ∨ lv_pt(PtrEndLoc).offset ≤ B.len)
        B.string = true;
        B.len = lv_pt(PtrEndLoc).offset;
/*} */
/* ensures *PtrEndText.is_nullt && */
        assert(B.string);
/*      *PtrEndText.strlen == 0 && */
        assert(B.len − rv_pt(PtrEndText).offset = 0);
/*      *PtrEndText.offset == ⌈*PtrEndText.offset⌉_entry + NbLine */
        assert(rv_pt(PtrEndText).offset = "*PtrEndText.offset".Pre + lv_pt(NbLine).val);
```

A report on the error in line [5] of main; the inequalities before execution of line [5] and a counter example:

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

(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)

Results

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

These are the results of running CSSV on extreme string manipulation procedures. As can be seen the outcome is very good, detects errors with low false alarm rate. But the consumed time and space resources are very high, increasing with procedure size. Time can be reduced by simplifying the constrains, as in Microsoft SAL which uses only {0,1} coefficients in the conditions.

In common cases where less complicated string manipulation is performed, the results are even better. Also one should have in mind that due to the fact that each procedure is analyzed alone, the memory consumption of the whole analysis is mostly influenced by the largest procedure.

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

Compile-Time Verification of Properties of Heap Intensive Programs by Mooly Sagiv, Thomas Reps, Reinhard Wilhem.

For more information you can look at:

- http://www.cs.tau.ac.il/~TVLA
- http://www.cs.tau.ac.il/~msagiv/toplas02.pdf

Introduction to Shape Analysis

The intention of this collaborative work is to verify through static analysis a code that makes intensive use of the heap.

The TVLA system was developed at Tel Aviv University. It produces static analysis of the code using set of rules given by the programmer. It handles specially issues and deals with dynamic allocations and pointers. In fact, TVLA does *Shape Analysis* of the code.

The purpose of Shape analysis is to determine the possible shape of a dynamically allocated data structure at a given program point, i.e., we want to prove some properties about pointers in the program. The motivation for such analysis is based on the fact that programs dealing with pointers and data structure allocated on the heap are more error prone.

As a compile time static analysis, shape analysis identifies complicated bugs at compile time, but it does more than this, it also automatically prove correctness!

Some of the relevant questions to shape analysis are:

- Does a pointer points to a shared element?
- Does a variable p points to an allocated element every time p is dereferenced, i.e., is our program trying to dereference a null pointer?
- Can a procedure cause a memory-leak?
- Does a variable point to an acyclic list?
- Does a variable point to a double linked list?

As you can see, some of the questions may be relevant to all the programs (as the first three), and some may be specific only to a small set of programs (as the last two). Shape analysis combines the two approaches: we can check general properties as specific properties that sometimes are relevant only to our program.

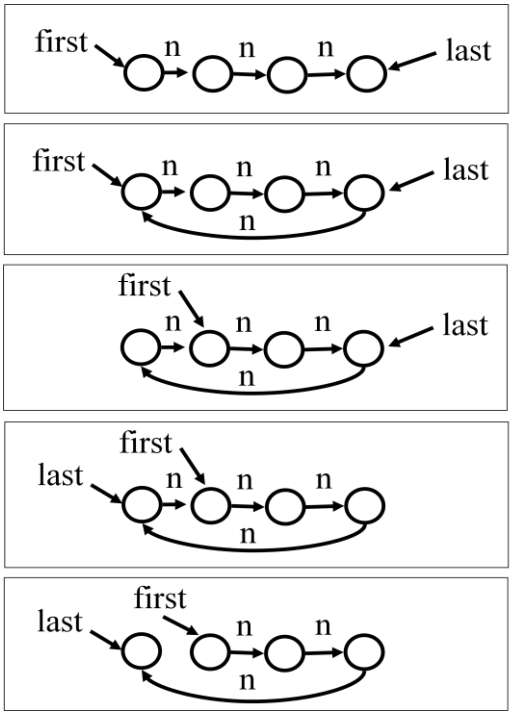
Some of the properties of heap manipulating programs which shape analysis can deal with are:

- No null dereference – for example, the Java JVM checks it at runtime, and if it finds one, it throws a NullPointerException.
- No memory leaks – in Java, the Garbage Collector is the responsible of cleaning all the unreferenced and not freed memory.

- Preservation of data structure invariant – we can define invariants which must be correct during the life of the program as first order logic predicates and the shape analysis proves them.
- Correct API usage – as in the previous one, we can define logic rules for verifying preconditions and post-conditions.
- Partial Correctness – shape analysis proves the correctness of each procedure alone, assuming that the preconditions are met in the entry to the procedure and proving that the set of invariants and the post-conditions are met at the exit of the procedure.
- Total Correctness – based on partial correctness shape analysis proves the total correctness of the program.

Let's look at an example of a function that rotates a linked list with a running example at its side:

```
rotate(List first, List last) {
      if ( first != NULL) {
            last → next = first;
            first = first → next;
            last = last → next;
            last → next = NULL;
      }
}
```



Note, that we started with an acyclic linked list, so that the data structure invariant is met at the entry of the procedure. During its run, the invariant is broken, but it is again met at the procedure exit. Static analysis should discover this and report it to the user.

Moreover, the shape analysis will report us that there are no null de-references and no memory leaks, as well as partially correctness hold (in order to prove partially correctness the programmer must specify assertions for the procedure entry and exit).

A word about partial correctness: even if the procedure contains recursion and other procedure calls which modify the heap and pointer values by themselves, our shape analysis will success proving the procedure correctness which yields the partial correctness of the code.

For example in the following quicksort code, it calls the partition method, which changes the value of the pointer p and q and the recursive call to the quicksort method which changes pointers as well as the list elements in order to sort the sub-lists.

```
List quickSort(List p, List q) {
     if(p==q || q == NULL)
          return p;
     List h = partition(p,q);
     List x = p→n;
     p →n = NULL;
     List low = quickSort(h, p);
     List high = quickSort(x, NULL);
     p→n = high;
     return low;
}
```

The challenges of shape analysis are to specify the program semantics and the desired properties as a set of logic rules (axioms, predicates) and to automatically verify them. The shape analysis should prove the desired properties using the program semantics. This is an undecidable problem even for simple programs and properties. Therefore TVLA uses a canonical heap abstraction which beyond of being conservative will allow us to prove partially correctness of the whole program.

Concrete Interpretation

Logical Structures (Labeled Graphs)

We will partially represent the concrete semantics. The concrete state will be an unbounded table of logical relations. The table will contain the following relations:

- Nullary relation symbols – represent boolean axioms in the program, such as that a variable x is positive.
- Unary relation symbols – represent the program reasoning, i.e. the properties of unbounded memory, as for example, for each variable we will remember where it points to.
- Binary relation symbols – represents the structures field relations (for example a next field in a list cell structure), these are in fact relations between memory locations.

We will use first order logic with transitive closure (FO^{TC}) over TC, $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$ to express logical structure properties (for example to express that our linked list is acyclic).

The concrete semantics will be composed of logical structures which provide meaning for the relations. There will be:

- A set of individuals (nodes) *U* – representing memory locations.
- Relation symbols *P*:
  - $p^0() \rightarrow \{0,1\}$ – representing nullary relations.
  - $p^1(u) \rightarrow \{0,1\}$ – representing unary relations – for each variable we will have such a relation p^1, which is 1 in the unique heap position pointed by this variable (if not pointing to null) and giving 0 to all the other positions. Similarly,
$$x(u) = \begin{cases} 1, & x\ points\ to\ u \\ 0, & otherwise \end{cases}$$
  - $p^2(u,v) \rightarrow \{0,1\}$ - representing binary relations – for each structure field f, if the field p^2 of the memory location u points to v, $p^2(u,v)$ will be 1, and for any memory location w different from v, $p^2(u,w) = 0$. Similarly,
$$n(u,v) = \begin{cases} 1, & u's\ n\ field\ points\ to\ v \\ 0, & otherwise \end{cases}$$

These relations will help us to cope with the problem that when defining the semantics of the language we don't know how much memory the program will allocate and which memory location will be allocated in each case.

These relations will also work well for variables pointing to the stack, not just to the heap.

Example – List creation:

We will use as an example a linked list given by:

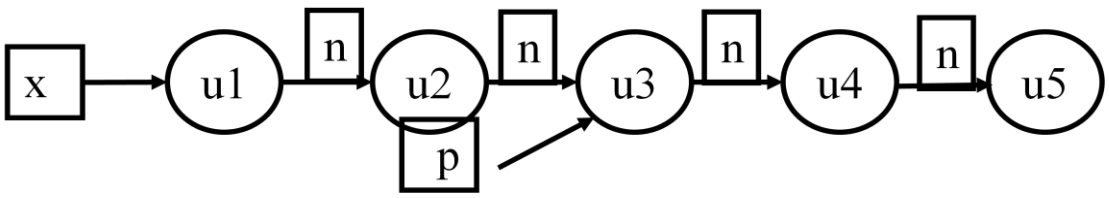
```
typedef struct node {
    int val;
    struct node *next;
} *List;
```

The respective relations for the following list will be:

– U = {u1, u2, u3, u4, u5}

– x = {u1}, p = {u3}

– n = {<u1, u2>, <u2, u3>, <u3, u4>, <u4, u5>}




```
List create (…)
```

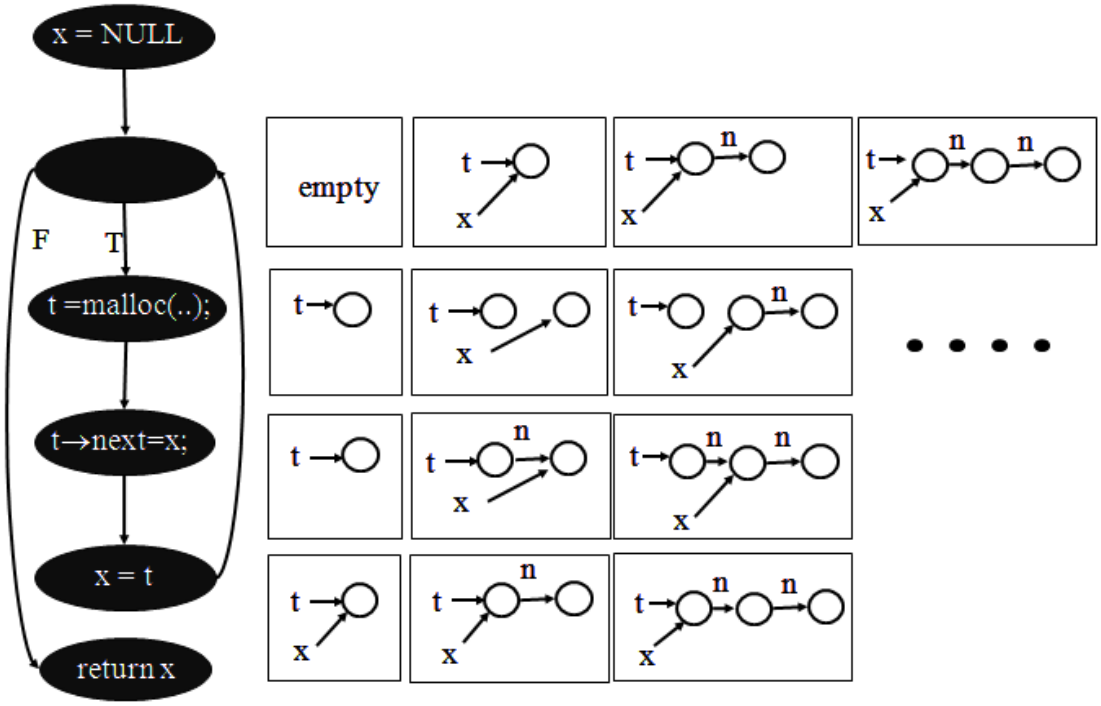
```
{
List x, t;
x = NULL;
while (…) do {
       t = malloc();
       t →next=x;
      x = t ;}
return x;
}
```

TVLA analysis will success showing that:

- There are no null dereferences.
- No memory leaks, i.e. there are no allocated memory locations which are not reachable from variables in the stack.
- The function returns an acyclic list.

Concrete Interpretation Example

Now, we will show the concrete interpretation for the list creation example:



In the example we can appreciate the control flow graph (CFG) of the create function. As expected, the concrete interpretation does not lose information about the pointer relationships in the function execution. We can see in each phase whether there are memory leaks or for example whether x and t point to the same memory location (in this case, list node).

Next we will present the concrete interpretation rules:

Statement	Update formula	Explanation

x =NULL	x'(v)= 0	x does not point to any location in the current state of the program
x= malloc()	x'(v) = IsNew(v)	This is a special tvla predicate, stating that x points to the new allocated memory location (there is only one new allocated)
x=y	x'(v)= y(v)	x now points to the same location that y points.
x=y →next	x'(v)= ∃w: y(w) ∧ n(w, v)	x points to v only if it is pointed by the next field of a memory location w which is pointed by y.
x →next=y	n'(v, w) = (¬x(v)∧ n(v, w)) ∨ (x(v) ∧ y(w))	In the first part of the "or" we cover all the memory location not pointed by x. in the second one we cover the one that is pointed by x.

Notes:

- In the updated formula column, x represents the "old value" and x' the "new value".
- In the x = y→next statement we also check null dereference! If Y points to null, there will be no such w sustaining the condition.
- The last updated formula can be written in a more readable way:

$$\boldsymbol{N}'(\boldsymbol{v},\boldsymbol{w}) = \begin{cases} \boldsymbol{N}(\boldsymbol{v},\boldsymbol{w}), & NOT(X(\boldsymbol{V})) \\ \boldsymbol{Y}(\boldsymbol{w}), & X(V) \end{cases}$$

We can also define invariants, this will be defined as first order logic predicates, and generally will deal with the general state of the program. Here are some examples:

Invariant	formula	Explanation
no garbage	∀v: $\vee_{\{x \in PVar\}}$ ∃w: x(w) ∧ n*(w, v)	True iff all the memory locations are reachable from some variable. N*(w,v) is the transitive closure, which indicates that v is pointed by 0 or more next field pointers.
acyclic list (x)	∀v, w: x(v) ∧ n*(v, w) → ¬n^+(w, v)	True iff there is no next field relation (v,w) reachable from x such that v is reachable from w.
reverse(x)	∀v, w, r: (x(v) ∧ n*(v, w) → n(w, r)) ↔ n'(r, w)	The ↔ represents a procedure since it defines for each two memory locations the next relation after applying the reverse procedure in terms of the previous relations before the procedure call.

Note that in the reverse invariant there is a potential threat of memory leak if the last element of the list is not reachable from another variable. The formula can be improved by adding a "last" relation, which indicates whether a location is the last node in a list. By this way we can check if the last node is reachable from a variable other than x.

Why do we use logical structures?

- Logical structures allow us to naturally model pointers and dynamic allocation.
- They also don't bind us on the number of memory location (unbounded memory).
- We can (relatively easily) use logical formulas to express semantics.
- Quantifiers allow us to represent store updates and the procedure concept (like we saw in the reverse example).
- Other models like concurrency and abstract fields (in Object Oriented Programming) can be modeled using logical structures.

Abstract Interpretation

Canonical Abstraction

For the abstract interpretation we will use 3-valued Logical Structures. The abstract semantics will be composed of logical structures which provide meaning for the relations. There will be:

- A set of individuals (nodes) U – representing memory locations.
- Relation symbols P:
  - $p^0() \rightarrow \{0,1,1/2\}$
  - $p^1(u) \rightarrow \{0,1,1/2\}$
  - $p^2(u,v) \rightarrow \{0,1,1/2\}$

About the ½ value:

- The relations are defined as before, except that now we add the ½ value.
- It will be like T (Top). We define a join semi-lattice: $0 \sqcup 1 = 1/2$
- The reason why we call it ½ is because it adapts to the concept of min (we think about it as logical or) and max (we think about it as logical and).

We will define a canonical abstraction function, β. We will define equivalence classes based on the values of the unary relations. In other words we will merge nodes which are pointed by the same set of variables. The equivalence classes which contain more than one node will be called **summary nodes.**

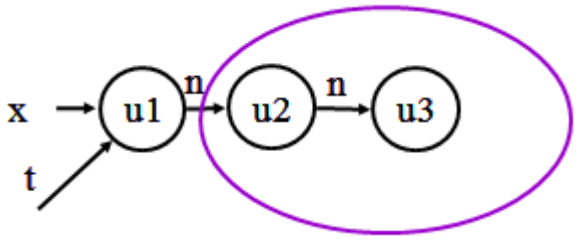
Now we define the abstract relations:

$$p^S(u'_1, ..., u'_k) = \sqcup \{p^B(u_1, ..., u_k) \mid f(u_1)=u'_1, ..., f(u_k)=u'_k\}$$

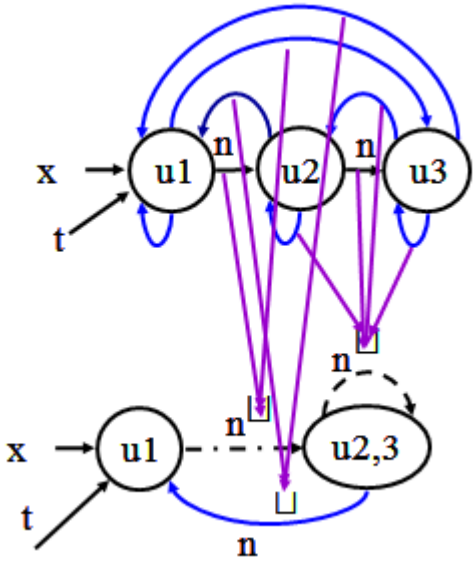
That is, we will define the unary relations of a merged set of locations as the join of all the unary relations of the original nodes.

If we have A memory location, they will yield to at most 2^A abstract individuals.

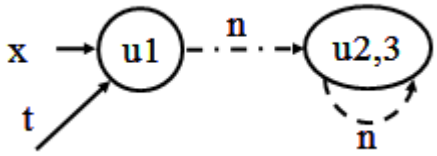
Example:



In this example there are 2 equivalence classes: u1 (pointed by x and t) and u2,u3 which are not directly pointed by any variable.



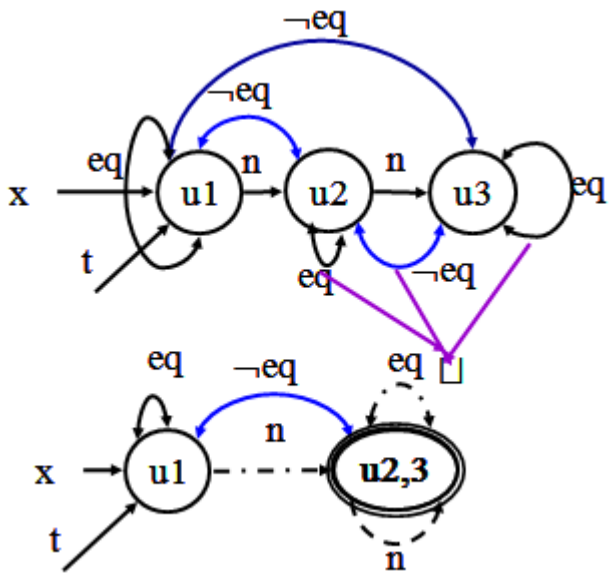
Now, we apply the join to each relation (in blue all the relation that are equal to 0). The dashed lines indicate ½ values. This are relations that in some cases are true (u1, u2) and in other false (u1,u3) and finally we get the following graph:



Canonical Abstraction and Equality

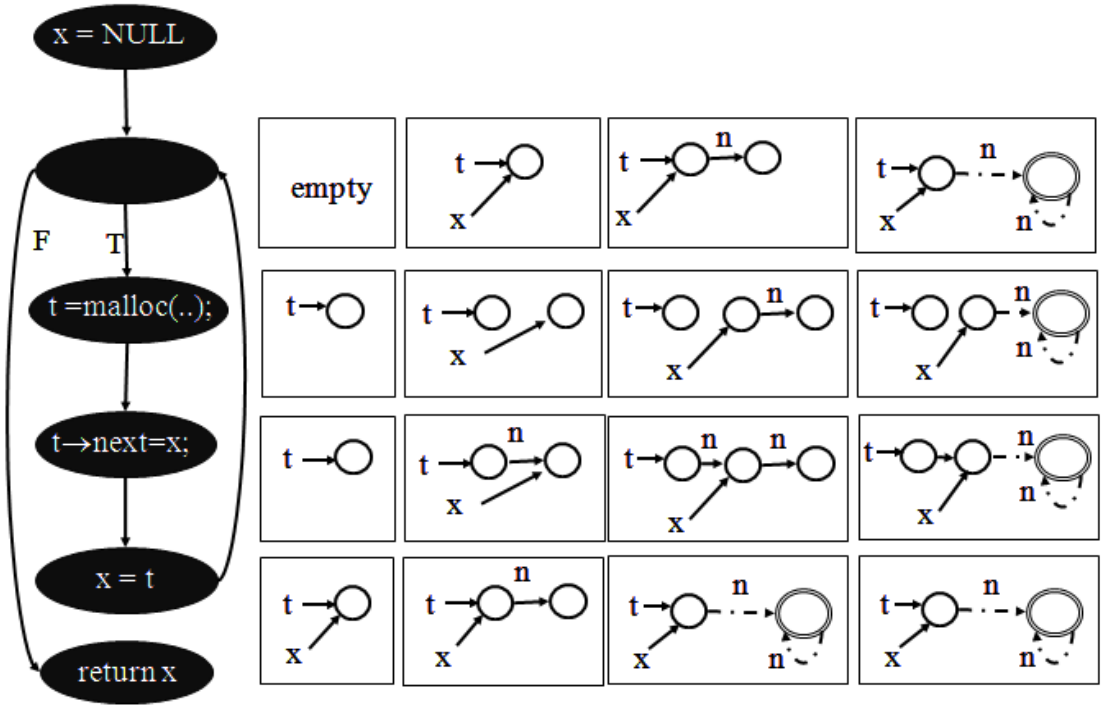
Since summary nodes may represent more than one element, in our abstraction we lose the information about the equality of locations and pointers. We need a predicate, *eq*, that will indicate the abstract interpretation whether this is a summary node or not. Summary nodes will be nodes with *eq(u,u)*=1/2 (since for each u,v such that u=v *eq*(u,v)=1, and for each u,v such that u≠v *eq*(u,v)=0).

The example with *eq* relations:



Abstract Interpretation Example

Now, we present the list creation example abstract interpretation:



We can observe that from the third iteration a summary node was created representing the tail of the list, in which no node is pointed by any variable. As we can see the value of the next relation between the first node and the summary node is ½, since the first node points only to the second. Also the next relation between the summary node and itself is ½ since each node points to the next one but not to the others.

Finally, we note that this shape analysis does the following abstractions:

- It ignores data types – the only thing that matters the analysis is pointer relationships.
- It ignores the values stored in the different variables or memory locations that are not pointers.

Another TVLA Example

We will analyze a small program that produces a memory leak. Next, we present the program code with the corresponding update logic statements.

1.	y = malloc(1);	y'(v) = IsNew(v)
2.	x = malloc(1);	x'(v) = IsNew(v)
3.	y = x;	y'(v) = x(v)

Let v_1 be the memory location allocated at line 1, and v_2 be the memory location allocated at line 2. Recall the "No Garbage" invariant: $\forall v: \vee_{\{x \in PVar\}} \exists w: x(w) \wedge n^*(w, v)$. Next, we will show how TVLA produces an error in this example:

Statement	$y(v_1)$	$y(v_2)$	$x(v_1)$	$x(v_2)$	Allocated locations	No Garbage
Initial state	False	False	False	False	$\varnothing$	True
y = malloc(1);	True	False	False	False	$\{v_1\}$	True
x = malloc(1);	True	False	False	True	$\{v_1, v_2\}$	True
y = x;	False	True	False	True	$\{v_1, v_2\}$	False

In blue, the updated predicates. In yellow, we show the predicates that give the error. As you can appreciate, after line 3, v_1 is not more reachable from any variable in the program.