Sendmail crackaddr - Static Analysis strikes back

Bogdan Mihaila

Technical University of Munich, Germany
June 15, 2014

Name Lastname < *name@mail.org* > ()()()()()()()()()() . . . ()()()

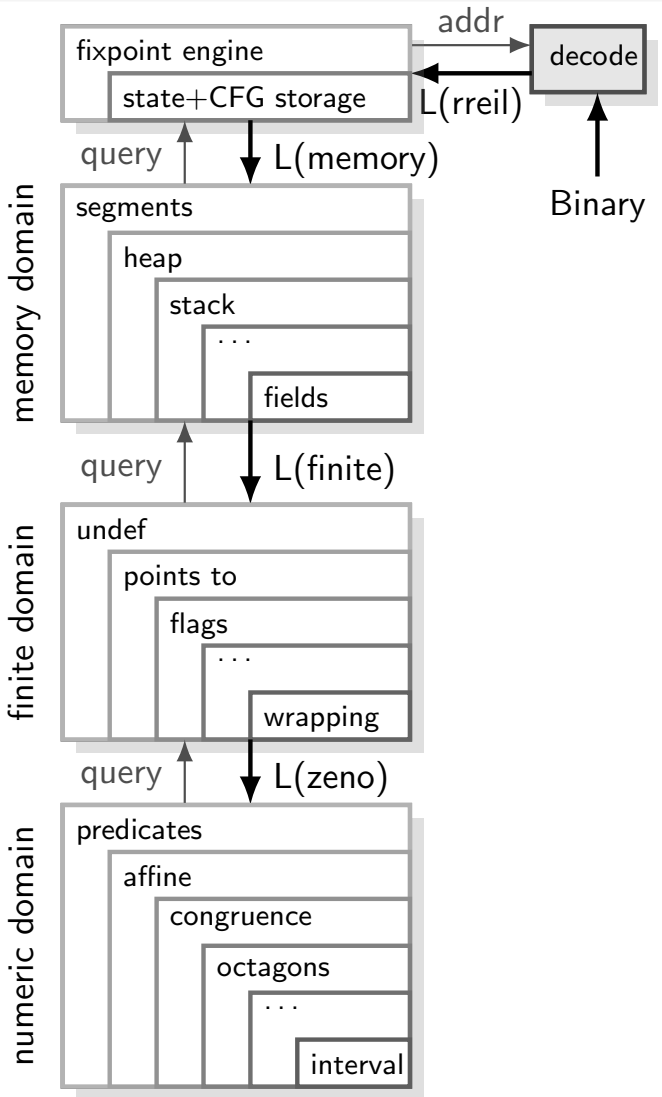
Analyzer Features

Static program analysis using abstract interpretation

- disassemble binaries (ELF, PE) in a recursive descent process
- perform a fixpoint computation to infer jump targets
- use abstract domains to infer memory bounds and flag out-of-bounds accesses
- can deal with overlapping instructions, unstructured control flow

Project page: `https://bitbucket.org/mihaila/bindead`

Analyzer Overview



- GDSL as disassembler frontend produces RREIL for the analysis
- RREIL gets transformed to simpler languages for the abstract domains
- modular construction using co-fibered abstract domains
- fixpoint and disassembly process are intertwined
- for interprocedural analysis we have a call-string and a summarization approach

Sendmail Bug

Discovered 2003 by Mark Dowd

Buffer overflow in an email address parsing function of Sendmail.
Consists of a parsing loop using a state machine. $\sim$500 LOC.

Sendmail Bug

Discovered 2003 by Mark Dowd

Buffer overflow in an email address parsing function of Sendmail. Consists of a parsing loop using a state machine. ∼500 LOC.

Thomas Dullien since then extracted a smaller version of the bug as an example for a hard problem for static analyzers. ∼50 LOC.

Sendmail Bug

Discovered 2003 by Mark Dowd

Buffer overflow in an email address parsing function of Sendmail. Consists of a parsing loop using a state machine. ∼500 LOC.

Thomas Dullien since then extracted a smaller version of the bug as an example for a hard problem for static analyzers. ∼50 LOC.

Why hard?

The join of states in abstract interpretation, widening, overapproximation introduce too much imprecision to be able to separate the states of the parsing automaton inside the loop. Thus the non-vulnerable version is flagged as vulnerable, too by a static analyzer.

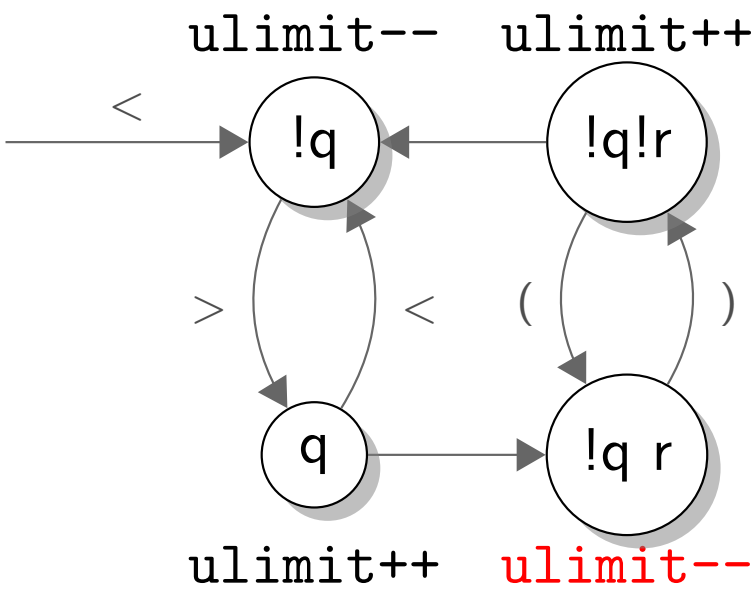
Sendmail Bug Code

```
1  #define BUFFERSIZE 200
2  #define TRUE 1
3  #define FALSE 0
4  int copy_it (char * input, unsigned int length) {
5      char c, localbuf[BUFFERSIZE];
6      unsigned int upperlimit = BUFFERSIZE - 10;
7      unsigned int quotation = roundquote = FALSE;
8      unsigned int inputIndex = outputIndex = 0;
9      while (inputIndex < length) {
10         c = input[inputIndex++];
11         if ((c == '<') && (!quotation)) {
12             quotation = TRUE; upperlimit--;
13         }
14         if ((c == '>') && (quotation)) {
15             quotation = FALSE; upperlimit++;
16         }
17         if ((c == '(') && (!quotation) && !roundquote) {
18             roundquote = TRUE; upperlimit--; // decrementation was missing in bug
19         }
20         if ((c == ')') && (!quotation) && roundquote) {
21             roundquote = FALSE; upperlimit++;
22         }
23         // If there is sufficient space in the buffer, write the character.
24         if (outputIndex < upperlimit) {
25             localbuf[outputIndex] = c; outputIndex++; }
26     }
27     if (roundquote) {
28         localbuf[outputIndex] = ')'; outputIndex++; }
29     if (quotation) {
30         localbuf[outputIndex] = '>'; outputIndex++; }
31 }
```

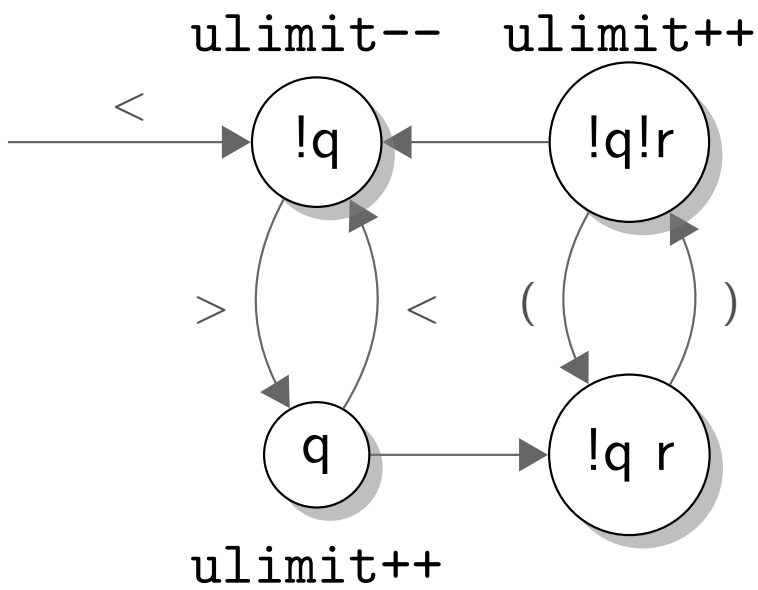
State machine of parser

We need to verify that `upperlimit < BUFFERSIZE` always holds in the good version and `upperlimit` is unbounded in the bad version.

Good:



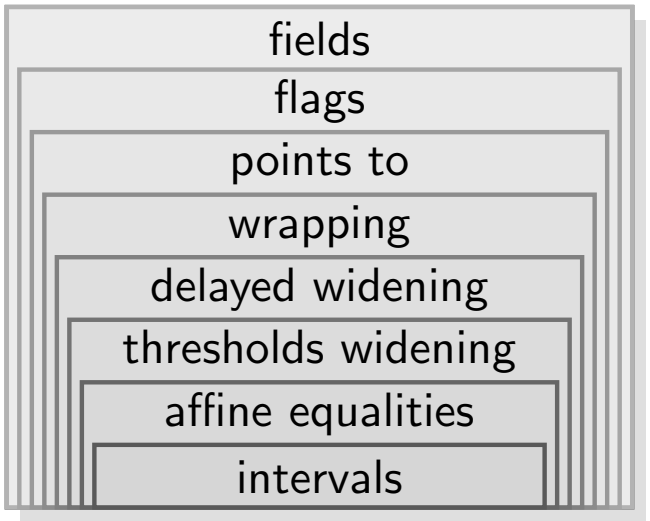
Bad:



In the bad version `upperlimit` can be steadily incremented and a write outside of the stack allocated buffer can be triggered.

Stack of used domains

To verify the code we used these abstract domains:



Adding more domains (e.g. predicates, congruences, intervalsets, octagons, polyhedra) improves the precision of the inferred bounds after widening but is not necessary to verify the invariant.

Analyzed Code

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {  upperlimit = 190, q = 0, rq = 0
9              q = 1; upperlimit--;
10         }    ⊔ : upperlimit + q = 190, upperlimit = [189, 190], q = [0, 1]
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {  upperlimit = 189, q = 1, rq = 0
12             q = 0; upperlimit++;
13         }    ⊔ : upperlimit + q = 190, upperlimit = [189, 190], q = [0, 1]
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {  upperlimit = 190, q = 0, rq = 0
15             rq = 1; upperlimit--;
16         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [189, 190], q = [0, 1], rq = [0, 1]
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {  upperlimit = 189, q = 0, rq = 1
18             rq = 0; upperlimit++;
19         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [189, 190], q = [0, 1], rq = [0, 1]
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) { upperlimit = [189, 190], outputIndex = 0
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments.
We analyze the loop again with the affine equality: $upperlimit + q + rq = 190$ still being holding

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) { widening ∇ becomes a join ⊔ due to constant flag assignments
7          c = input[inputIndex++];
8          if (... && (!q)) { upperlimit + rq = 190, upperlimit = [189, 190], q = 0, rq = [0, 1]
9              q = 1; upperlimit--;
10         }   ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments.
We analyze the loop again with the affine equality: $upperlimit + q + rq = 190$ still being holding

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {  upperlimit + rq = 189, upperlimit = [188, 189], q = 1, rq = [0, 1]
12             q = 0; upperlimit++;
13         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments.
We analyze the loop again with the affine equality: $upperlimit + q + rq = 190$ still being holding

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {  upperlimit = 190, q = 0, rq = 0
15             rq = 1; upperlimit--;
16         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments.
We analyze the loop again with the affine equality: $upperlimit + q + rq = 190$ still being holding

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {  upperlimit = 189, q = 0, rq = 1
18             rq = 0; upperlimit++;
19         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments.
We analyze the loop again with the affine equality: $upperlimit + q + rq = 190$ still being holding

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) { upperlimit = [188, 190], outputIndex = [0, 1]
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

3rd iteration: now widening ∇ is applied and we restrict *outputIndex* to $\mathit{outputIndex} < \mathit{upperlimit}$.
As *upperlimit* did not change from the previous iteration it is not affected by widening.

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) { ∇ : outputIndex = [0, 189]
7          c = input[inputIndex++];
8          if (... && (!q)) { upperlimit + rq = 190, upperlimit = [189, 190], q = 0, rq = [0, 1]
9              q = 1; upperlimit--;
10         }   ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

3rd iteration: now widening ∇ is applied and we restrict *outputIndex* to $outputIndex < upperlimit$.
As *upperlimit* did not change from the previous iteration it is not affected by widening.

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {  upperlimit + rq = 189, upperlimit = [188, 189], q = 1, rq = [0, 1]
12             q = 0; upperlimit++;
13         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

3rd iteration: now widening ∇ is applied and we restrict *outputIndex* to $outputIndex < upperlimit$.
As *upperlimit* did not change from the previous iteration it is not affected by widening.

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {  upperlimit = 190, q = 0, rq = 0
15             rq = 1; upperlimit--;
16         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

3rd iteration: now widening ∇ is applied and we restrict *outputIndex* to $outputIndex < upperlimit$.
As *upperlimit* did not change from the previous iteration it is not affected by widening.

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {  upperlimit = 189, q = 0, rq = 1
18             rq = 0; upperlimit++;
19         }    ⊔ : upperlimit + q + rq = 190, upperlimit = [188, 190], q = [0, 1], rq = [0, 1]
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

3rd iteration: now widening ∇ is applied and we restrict *outputIndex* to $outputIndex < upperlimit$.
As *upperlimit* did not change from the previous iteration it is not affected by widening.

```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) { upperlimit = [188, 190], outputIndex = [0, 189]
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) {
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Analyzed Code

4th iteration: loop is stable; outside of the loop body the value of *outputIndex* is still bounded

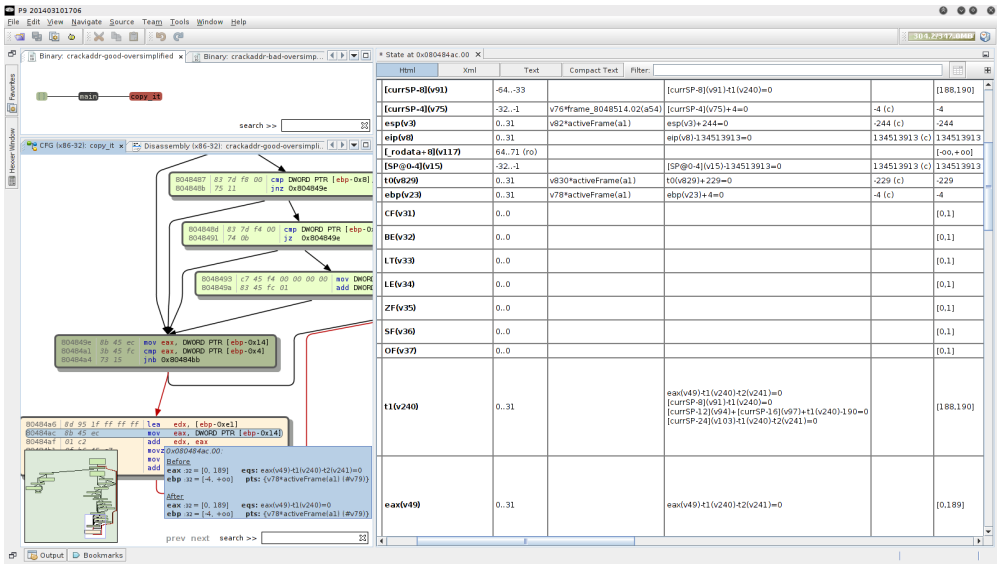
```
1  int copy_it (char * input, unsigned int length) {
2      char c, localbuf[200];
3      unsigned int upperlimit = 190;
4      unsigned int q = rq = 0;
5      unsigned int inputIndex = outputIndex = 0;
6      while (inputIndex < length) {
7          c = input[inputIndex++];
8          if (... && (!q)) {
9              q = 1; upperlimit--;
10         }
11         if (... && (q)) {
12             q = 0; upperlimit++;
13         }
14         if (... && (!q) && !rq) {
15             rq = 1; upperlimit--;
16         }
17         if (... && (!q) && rq) {
18             rq = 0; upperlimit++;
19         }
20         if (outputIndex < upperlimit) {
21             localbuf[outputIndex] = c; outputIndex++; }
22     }
23     if (rq) { outputIndex = [0, 190]
24         localbuf[outputIndex] = ')'; outputIndex++; }
25     if (q) {  outputIndex = [1, 191]
26         localbuf[outputIndex] = '>'; outputIndex++; }
27 }
```

Key Points

- widening needs to be suppressed until the flag variables are stable to be able to infer the equality relation with *upperlimit*
- the inferred equality $upperlimit + q + rq = 190$ keeps *upperlimit* reduced inside the state machine
  $\rightsquigarrow$ the possible values of *upperlimit* are restricted inside the loop and thus widening will not introduce spurious values
- the threshold $outputIndex < upperlimit$ from line 20 must be used when widening *outputIndex*
  $\rightsquigarrow$ *outputIndex* is also restricted outside of the loop for the next two writes
- in the vulnerable version because of the missing decrementation the equality relation does not hold
  $\rightsquigarrow$ *upperlimit* is unbounded after widening

Demo

Time to prove it!



Unfortunately

The original code is much more complex!

Has ∼10 loops (nesting depth is 4), gotos, lots of pointer arithmetic, calls to string functions . . .

Unfortunately

The original code is much more complex!

Has ∼10 loops (nesting depth is 4), gotos, lots of pointer arithmetic, calls to string functions . . .

$\rightsquigarrow$ we cannot yet analyze that

Conclusion

- abstract domains can infer surprisingly nice results
  in this case a simpler invariant than expected
- but the fixpoint computation and the reduction between the domains is quite complex
  $\leadsto$ it is hard to debug and reason about the inferred invariants given the load of data
- if an expected invariant cannot be proved it is difficult to find out why and fix it
- being able to understand and debug an abstract analyzer is key to building useful analyses!