

# Sendmail crackaddr - Static Analysis strikes back

Bogdan Mihaila

Technical University of Munich, Germany  
December 6, 2014

Name Lastname < *name@mail.org* > (0000000000) ... (000)

## Static program analysis using abstract interpretation

- use abstract domains to over-approximate concrete states
- abstract transformers simulate the concrete program semantics on the abstract state
- perform a fixpoint computation to infer invariants for each program point
- merge over all paths over-approximates all possible program executions (soundness)
- precision depends on the abstraction (completeness)
- for termination widening is necessary (introduces imprecision)

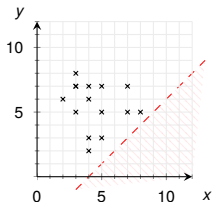
# Abstraction Examples

Some examples of concrete values and their abstractions ...

# Sets of Concrete Values and their Abstractions

Concrete Points

$$\pm x = c$$

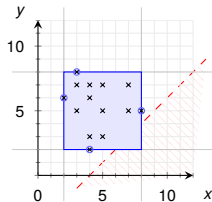


Constraints:

$$\begin{aligned} &x = 2 \wedge y = 6 \\ &\vee x = 3 \wedge y = 5 \\ &\vee x = 3 \wedge y = 7 \\ &\vee x = 3 \wedge y = 8 \\ &\vee \dots \end{aligned}$$

Intervals

$$\pm x \leq c$$

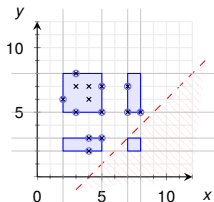


Constraints:

$$\begin{aligned} &2 \leq x \wedge x \leq 8 \\ &\wedge 2 \leq y \wedge y \leq 8 \end{aligned}$$

Interval Sets

$$\bigvee_j (l_j \leq x \wedge x \leq u_j)$$

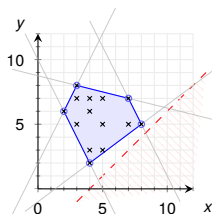


Constraints:

$$\begin{aligned} &2 \leq x \wedge x \leq 5 \\ &\vee 7 \leq x \wedge x \leq 8 \\ &\vee 2 \leq y \wedge y \leq 3 \\ &\vee 5 \leq y \wedge y \leq 8 \end{aligned}$$

Polyhedra

$$\sum_i a_i x_i \leq c$$



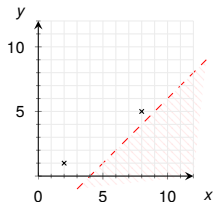
Constraints:

$$\begin{aligned} &2x - y \leq -2 \\ &\wedge -2x - y \leq -10 \\ &\wedge 2x + y \leq -21 \\ &\wedge 3x + 4y \leq 4 \\ &\wedge x + 4y \leq 35 \end{aligned}$$

# Sets of Concrete Values and their Abstractions

Concrete Points

$$\pm x = c$$

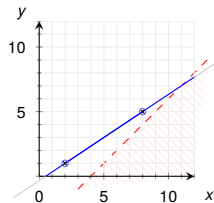


Constraints:

$$\begin{aligned} x &= 2 \wedge y = 1 \\ \vee x &= 8 \wedge y = 5 \end{aligned}$$

Affine Equalities

$$\sum_i a_i x_i = c$$

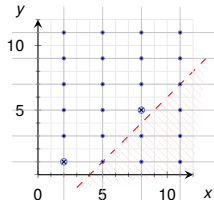


Constraints:

$$2x - 3y = 3$$

Congruences

$$x \equiv b \pmod{a}$$



Constraints:

$$\begin{aligned} x &\equiv 2 \pmod{3} \\ \wedge y &\equiv 1 \pmod{2} \end{aligned}$$

# Operations on Abstractions

Some examples of operations on abstractions ...

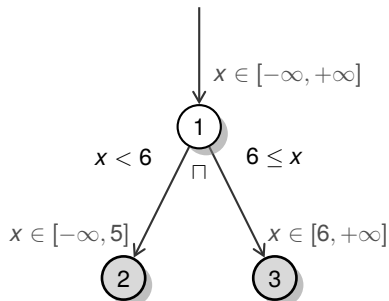
# Some Operations on Intervals

Arithmetics:

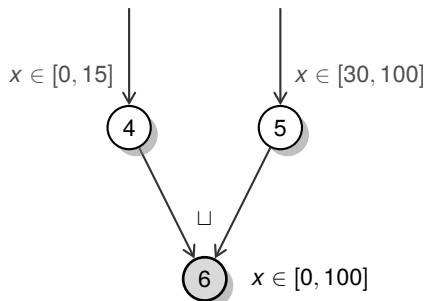
$$[0, 100] + [1, 2] = [1, 102]$$

$$[0, 100] - [1, 2] = [-2, 99]$$

Tests or Assumptions, Meet  $\sqcap$



Merge of paths, Join  $\sqcup$



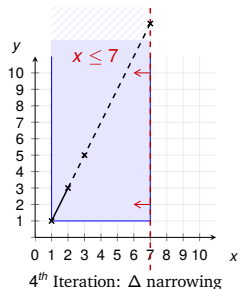
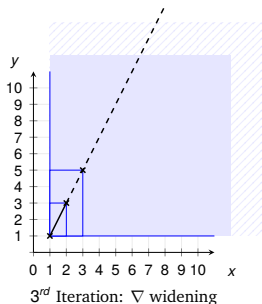
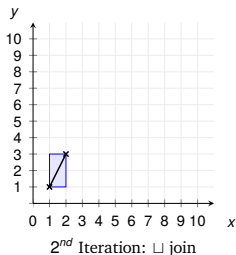
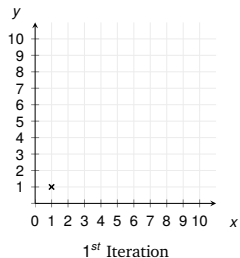
## Widening and Narrowing

To analyze loops in less steps than the real iterations count ...  
and especially always analyze loops in finite steps.  
Analysis Termination!



# Widening and Narrowing on Intervals

```
int x = 1;
int y = 1;
// shown x, y values
// are at loop head
while (x <= 6) {
    x = x + 1;
    y = y + 2;
}
```



# Abstract Interpretation

## Good intro and overview material:

- P. Cousot and R. Cousot, **A gentle introduction to formal verification of computer systems by abstract interpretation**, 2010
- P. Cousot, **Abstract Interpretation Based Formal Methods and Future Challenges**, 2001
- P. Cousot and R. Cousot, **Abstract Interpretation: Past, Present and Future**, 2014

Now to the Analyzer ...

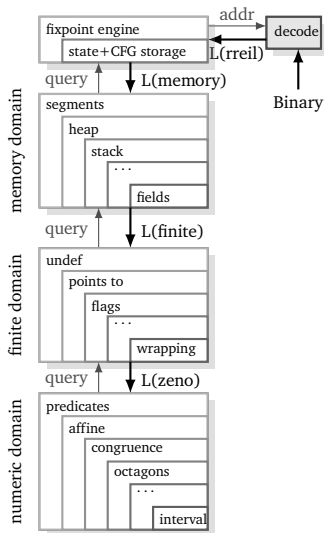
# Analyzer Features

## Analysis of binaries using abstract interpretation

- analyze machine code from disassembled executables
- translate machine code to intermediate language (RREIL)
- abstract transformers for instruction semantics of RREIL
- perform a reachability analysis to infer jump targets and
- use abstract domains to infer memory bounds and flag out-of-bounds accesses

Project page: <https://bitbucket.org/mihaila/bindead>

# Analyzer Overview



- disassembler frontend produces RREIL for the analysis
- RREIL gets transformed to simpler languages for the abstract domains
- fixpoint and disassembly process are intertwined
- modular construction using co-fibered abstract domains
- domains stack is a partially reduced product of domains
- for interprocedural analysis we use either call-string or a summarization approach

And finally to Sendmail ...

# 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.

# 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.

## Bounty for Static Analyzers since 2011

Thomas Dullien extracted a smaller version of the bug as an example of 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.

## Bounty for Static Analyzers since 2011

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

## Since then . . .

Various talks at security conferences about the static analysis of the problem and a paper using Havoc/Boogie. Unfortunately, their solution required manual specification of the loop invariant.

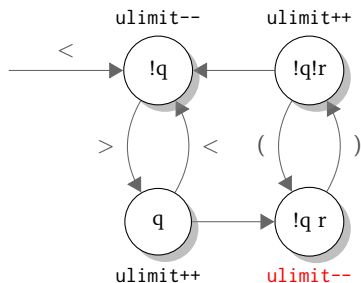
# Sendmail Bug Code

```
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;
26            outputIndex++;
27        }
28    }
29    if (roundquote) {
30        localbuf[outputIndex] = ')'; outputIndex++; }
31    if (quotation) {
32        localbuf[outputIndex] = '>'; outputIndex++; }
33 }
```

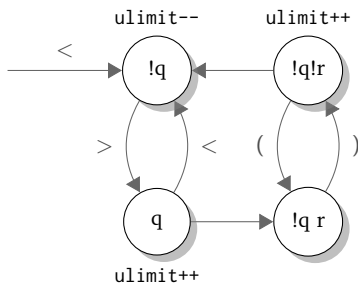
# State Machine of Parser

We need to verify that  $\text{outputIndex} < \text{upperlimit} < \text{BUFFERSIZE}$  always holds in the good version.

Good:



Bad:



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

# Sendmail Bug Analysis

Why hard?

**Symbolic Execution**

Runs into path explosion due to the loop and automaton.

# Sendmail Bug Analysis

Why hard?

## Symbolic Execution

Runs into path explosion due to the loop and automaton.

## Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

# Sendmail Bug Analysis

Why hard?

## Symbolic Execution

Runs into path explosion due to the loop and automaton.

## Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

## Abstract Interpretation

The over-approximation, the join and widening of states introduce too much imprecision. Not able to separate the states of the parsing automaton inside the loop.

# Sendmail Bug Analysis

Why hard?

## Symbolic Execution

Runs into path explosion due to the loop and automaton.

## Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

## Abstract Interpretation

The over-approximation, the join and widening of states introduce too much imprecision. Not able to separate the states of the parsing automaton inside the loop.

# Sendmail Bug Analysis

Why hard?

## Symbolic Execution

Runs into path explosion due to the loop and automaton.

## Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

## Abstract Interpretation

The over-approximation, the join and widening of states introduce too much imprecision. Not able to separate the states of the parsing automaton inside the loop.

↪ the non-vulnerable version is flagged as vulnerable, too, by a static analyzer



# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0    prove memory correctness for all possible concrete inputs!
int copy_it (char *input, unsigned int length) { *input[i] ∈ [-∞, +∞], length ∈ [0, +∞]
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++;
        }
    }
    if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0; inputIndex ∈ [0, 0], outputIndex ∈ [0, 0]
    while (inputIndex < length) {
        c = input[inputIndex++]; inputIndex ∈ [1, 1]
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++; outputIndex ∈ [1, 1]
             $\sqcup$ : outputIndex ∈ [0, 1]
        }
        if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
            localbuf[outputIndex] = ')'; outputIndex++; }
        if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
            localbuf[outputIndex] = '>'; outputIndex++; }
    }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0; inputIndex ∈ [0, 0], outputIndex ∈ [0, 0]
    while (inputIndex < length) { widening ∇: inputIndex ∈ [0, +∞], outputIndex ∈ [0, +∞]
        c = input[inputIndex++]; inputIndex ∈ [1, 1]
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++; outputIndex ∈ [1, 1]
        } □: outputIndex ∈ [0, 1]
    }
    if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0; inputIndex ∈ [0, 0], outputIndex ∈ [0, 0]
    while (inputIndex < length) { widening ∇: inputIndex ∈ [0, +∞], outputIndex ∈ [0, +∞]
        c = input[inputIndex++]; inputIndex ∈ [1, 1]
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) { use threshold outputIndex < upperlimit for widening!
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++; outputIndex ∈ [1, 1]
        } □: outputIndex ∈ [0, 1]
    }
    if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } □ : upperlimit ∈ [189, 190]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } □ : upperlimit ∈ [189, 191]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } □ : upperlimit ∈ [188, 191]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } □ : upperlimit ∈ [188, 192]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { widening ▽ removes bounds: upperlimit ∈  $[-\infty, +\infty]$ 
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } □: upperlimit ∈ [189, 190]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } □: upperlimit ∈ [189, 191]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } □: upperlimit ∈ [188, 191]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } □: upperlimit ∈ [188, 192]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { widening ∇ removes bounds: upperlimit ∈ [-∞, +∞]
        c = input[inputIndex++]; use relation with flag variables quotation and roundquote
        if ((c == '<') && (!quotation)) { to keep upperlimit bounded!
            quotation = TRUE; upperlimit--;
        } □: upperlimit ∈ [189, 190]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } □: upperlimit ∈ [189, 191]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } □: upperlimit ∈ [188, 191]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } □: upperlimit ∈ [188, 192]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE; quotation ∈ [0, 0], roundquote ∈ [0, 0]
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } ⊥ : quotation ∈ [0, 1]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } ⊥ : quotation ∈ [0, 1]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } ⊥ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } ⊥ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```



# Sendmail Code Revisited (Problems and Ideas)

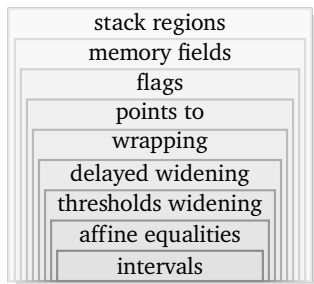
```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE; quotation ∈ [0, 0], roundquote ∈ [0, 0]
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { ∇ removes bounds: quotation ∈ [0, +∞], roundquote ∈ [0, +∞]
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } □: quotation ∈ [0, 1]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } □: quotation ∈ [0, 1]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } □: quotation ∈ [0, 1], roundquote ∈ [0, 1]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } □: quotation ∈ [0, 1], roundquote ∈ [0, 1]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE; quotation ∈ [0, 0], roundquote ∈ [0, 0]
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { ∇ removes bounds: quotation ∈ [0, +∞], roundquote ∈ [0, +∞]
        c = input[inputIndex++]; delay widening until flags and relations stable!
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } □ : quotation ∈ [0, 1]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } □ : quotation ∈ [0, 1]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } □ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } □ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Stack of required domains

To verify the code (disassembled from the binary) we used these abstract domains:



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

# Analyzed Code

Lets analyze the code!

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

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

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { inputIndex = 0, outputIndex = 0, length ∈ [-∞, +∞]
        c = input[inputIndex++];
        if (... && (!q)) { upperlimit = 190, q = 0, rq = 0
            q = 1; upperlimit--;
        } ⊢ : upperlimit + q = 190, upperlimit ∈ [189, 190], q ∈ [0, 1]
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

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

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) { upperlimit = 189, q = 1, rq = 0
            q = 0; upperlimit++;
        }  $\sqcup$  : upperlimit + q = 190, upperlimit  $\in$  [189, 190], q  $\in$  [0, 1]
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

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

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) { upperlimit = 190, q = 0, rq = 0
            rq = TRUE; upperlimit--;
        }  $\sqcup$  : upperlimit + q + rq = 190, upperlimit  $\in$  [189, 190], q  $\in$  [0, 1], rq  $\in$  [0, 1]
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

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

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) { upperlimit = 189, q = 0, rq = 1
            rq = 0; upperlimit++;
        }  $\sqcup$  : upperlimit + q + rq = 190, upperlimit  $\in$  [189, 190], q  $\in$  [0, 1], rq  $\in$  [0, 1]
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```



# Analyzed Code

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

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) { upperlimit ∈ [189, 190], outputIndex = 0
            localbuf[outputIndex] = c;
            outputIndex++;
        } □ : upperlimit ∈ [189, 190], outputIndex ∈ [0, 1]
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

2nd iteration: widening  $\nabla$  suppressed by the “delayed widening” domain because of the flag assignments.  
Join  $\sqcup$  performed instead. We analyze the loop again with still valid equality:  $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {  $\sqcup : inputIndex \in [0, 1], outputIndex \in [0, 1]$ 
        c = input[inputIndex++];
        if (... && (!q)) {  $upperlimit + rq = 190, upperlimit \in [189, 190], q = 0, rq \in [0, 1]$ 
            q = 1; upperlimit--;
        }  $\sqcup : upperlimit + q + rq = 190, upperlimit \in [188, 190], q \in [0, 1], rq \in [0, 1]$ 
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

2nd iteration: widening  $\nabla$  suppressed by the “delayed widening” domain because of the flag assignments.  
Join  $\sqcup$  performed instead. We analyze the loop again with still valid equality:  $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) { upperlimit + rq = 189, upperlimit ∈ [188, 189], q = 1, rq ∈ [0, 1]
            q = 0; upperlimit++;
        }  $\sqcup$  : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

2nd iteration: widening  $\nabla$  suppressed by the “delayed widening” domain because of the flag assignments.  
Join  $\sqcup$  performed instead. We analyze the loop again with still valid equality:  $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) { upperlimit = 190, q = 0, rq = 0
            rq = TRUE; upperlimit--;
        }  $\sqcup$  : upperlimit + q + rq = 190, upperlimit  $\in$  [188, 190], q  $\in$  [0, 1], rq  $\in$  [0, 1]
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

2nd iteration: widening  $\nabla$  suppressed by the “delayed widening” domain because of the flag assignments. Join  $\sqcup$  performed instead. We analyze the loop again with still valid equality:  $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) { upperlimit = 189, q = 0, rq = 1
            rq = 0; upperlimit++;
        }  $\sqcup : upperlimit + q + rq = 190, upperlimit \in [188, 190], q \in [0, 1], rq \in [0, 1]$ 
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

2nd iteration: widening  $\nabla$  suppressed by the “delayed widening” domain because of the flag assignments.  
Join  $\sqcup$  performed instead. We analyze the loop again with still valid equality:  $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) { upperlimit ∈ [188, 190], outputIndex ∈ [0, 1]
            localbuf[outputIndex] = c;
            outputIndex++;
        }  $\sqcup$  : upperlimit ∈ [188, 190], outputIndex ∈ [0, 2]
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

3rd iteration: now widening  $\nabla$  is applied using the widening threshold:  $outputIndex - 1 < upperlimit$ .  
Widening changes the lower bound of  $upperlimit$  but reduction with the equality  $upperlimit + q + rq = 190$  restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {  $\nabla$  :  $inputIndex \in [0, +\infty], outputIndex \in [0, 190], upperlimit \in [0, 190]$ 
        c = input[inputIndex++];
        if (... && (!q)) {  $upperlimit + rq = 190, upperlimit \in [189, 190], q = 0, rq \in [0, 1]$ 
            q = 1; upperlimit--;
        }  $\sqcup$  :  $upperlimit + q + rq = 190, upperlimit \in [188, 190], q \in [0, 1], rq \in [0, 1]$ 
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

3rd iteration: now widening  $\nabla$  is applied using the widening threshold:  $outputIndex - 1 < upperlimit$ .  
Widening changes the lower bound of  $upperlimit$  but reduction with the equality  $upperlimit + q + rq = 190$  restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) { upperlimit + rq = 189, upperlimit ∈ [188, 189], q = 1, rq ∈ [0, 1]
            q = 0; upperlimit++;
        }  $\sqcup$  : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```



# Analyzed Code

3rd iteration: now widening  $\nabla$  is applied using the widening threshold:  $outputIndex - 1 < upperlimit$ .  
Widening changes the lower bound of  $upperlimit$  but reduction with the equality  $upperlimit + q + rq = 190$  restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) { upperlimit = 190, q = 0, rq = 0
            rq = TRUE; upperlimit--;
        } □ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

3rd iteration: now widening  $\nabla$  is applied using the widening threshold:  $outputIndex - 1 < upperlimit$ .  
Widening changes the lower bound of  $upperlimit$  but reduction with the equality  $upperlimit + q + rq = 190$  restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) { upperlimit = 189, q = 0, rq = 1
            rq = 0; upperlimit++;
        }  $\sqcup : upperlimit + q + rq = 190, upperlimit \in [188, 190], q \in [0, 1], rq \in [0, 1]$ 
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

3rd iteration: now widening  $\nabla$  is applied using the widening threshold:  $outputIndex - 1 < upperlimit$ .  
Widening changes the lower bound of  $upperlimit$  but reduction with the equality  $upperlimit + q + rq = 190$  restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) { upperlimit ∈ [188, 190], outputIndex ∈ [0, 189]
            localbuf[outputIndex] = c;
            outputIndex++;
        } □ : upperlimit ∈ [188, 190], outputIndex ∈ [0, 190]
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Analyzed Code

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

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {  $\square$ :  $inputIndex \in [0, +\infty]$ ,  $outputIndex \in [0, 190]$ 
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {  $outputIndex \in [0, 190]$ 
        localbuf[outputIndex] = '>'; outputIndex++; }
    if (q) {  $outputIndex \in [1, 191]$ 
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

# Key Points

- widening needs to be suppressed until the flag variables are stable to infer the equality relation with *upperlimit*
- the inferred equality  $upperlimit + q + rq = 190$  and reduction between domains results in more precise values for *upperlimit*; recovers precision loss of widening
- narrowing does not help here; instead the threshold  $outputIndex < upperlimit$  must be used for widening  
     $\rightsquigarrow$  *outputIndex* is also restricted outside of the loop for the following two writes to the buffer
- in the vulnerable version because of the missing decrementation the equality relation does not hold  
     $\rightsquigarrow$  *upperlimit* is unbounded after widening

# Unfortunately

**The original Sendmail code is more complex!**

- the fix is not only one line but in more than just one place

# Unfortunately

## The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ~10 loops (nesting depth is 4) and gotos

# Unfortunately

## The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ~10 loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops



# Unfortunately

## The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ~10 loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops
- uses string manipulating functions

# Unfortunately

## The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ~10 loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops
- uses string manipulating functions

# Unfortunately

## The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains  $\sim 10$  loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops
- uses string manipulating functions

$\leadsto$  we cannot yet prove the invariant on that code

# Conclusion

- abstract domains can infer surprisingly nice results.  
     $\leadsto$  in this case a simpler invariant than expected.

# Conclusion

- abstract domains can infer surprisingly nice results.  
     $\leadsto$  in this case a simpler invariant than expected.
- but the fixpoint computation and the reduction between the domains is quite complex  
     $\leadsto$  hard to debug and reason about inferred invariants given the load of data (multiply with verbosity of machine code)

# Conclusion

- abstract domains can infer surprisingly nice results.  
     $\leadsto$  in this case a simpler invariant than expected.
- but the fixpoint computation and the reduction between the domains is quite complex  
     $\leadsto$  hard to debug and reason about inferred invariants given the load of data (multiply with verbosity of machine code)
- if an expected invariant cannot be proved it is difficult to find out why and fix it

# Conclusion

- abstract domains can infer surprisingly nice results.  
     $\leadsto$  in this case a simpler invariant than expected.
- but the fixpoint computation and the reduction between the domains is quite complex  
     $\leadsto$  hard to debug and reason about inferred invariants given the load of data (multiply with verbosity of machine code)
- 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!  
     $\leadsto$  general adoption of static analyzers is an uphill battle :(

# Time to analyze it!

The screenshot displays a debugger window with a disassembly view on the right and a call stack on the left. The disassembly view shows the following instructions:

Address	Disassembly	Comment	Flags
00401000	main		
00401001	push ebp		
00401002	mov ebp, esp		
00401003	push esi		
00401004	push edi		
00401005	push ebx		
00401006	push ecx		
00401007	push edx		
00401008	push esi		
00401009	push edi		
0040100A	push ebx		
0040100B	push ecx		
0040100C	push edx		
0040100D	push esi		
0040100E	push edi		
0040100F	push ebx		
00401010	push ecx		
00401011	push edx		
00401012	push esi		
00401013	push edi		
00401014	push ebx		
00401015	push ecx		
00401016	push edx		
00401017	push esi		
00401018	push edi		
00401019	push ebx		
0040101A	push ecx		
0040101B	push edx		
0040101C	push esi		
0040101D	push edi		
0040101E	push ebx		
0040101F	push ecx		
00401020	push edx		
00401021	push esi		
00401022	push edi		
00401023	push ebx		
00401024	push ecx		
00401025	push edx		
00401026	push esi		
00401027	push edi		
00401028	push ebx		
00401029	push ecx		
0040102A	push edx		
0040102B	push esi		
0040102C	push edi		
0040102D	push ebx		
0040102E	push ecx		
0040102F	push edx		
00401030	push esi		
00401031	push edi		
00401032	push ebx		
00401033	push ecx		
00401034	push edx		
00401035	push esi		
00401036	push edi		
00401037	push ebx		
00401038	push ecx		
00401039	push edx		
0040103A	push esi		
0040103B	push edi		
0040103C	push ebx		
0040103D	push ecx		
0040103E	push edx		
0040103F	push esi		
00401040	push edi		
00401041	push ebx		
00401042	push ecx		
00401043	push edx		
00401044	push esi		
00401045	push edi		
00401046	push ebx		
00401047	push ecx		
00401048	push edx		
00401049	push esi		
0040104A	push edi		
0040104B	push ebx		
0040104C	push ecx		
0040104D	push edx		
0040104E	push esi		
0040104F	push edi		
00401050	push ebx		
00401051	push ecx		
00401052	push edx		
00401053	push esi		
00401054	push edi		
00401055	push ebx		
00401056	push ecx		
00401057	push edx		
00401058	push esi		
00401059	push edi		
0040105A	push ebx		
0040105B	push ecx		
0040105C	push edx		
0040105D	push esi		
0040105E	push edi		
0040105F	push ebx		
00401060	push ecx		
00401061	push edx		
00401062	push esi		
00401063	push edi		
00401064	push ebx		
00401065	push ecx		
00401066	push edx		
00401067	push esi		
00401068	push edi		
00401069	push ebx		
0040106A	push ecx		
0040106B	push edx		
0040106C	push esi		
0040106D	push edi		
0040106E	push ebx		
0040106F	push ecx		
00401070	push edx		
00401071	push esi		
00401072	push edi		
00401073	push ebx		
00401074	push ecx		
00401075	push edx		
00401076	push esi		
00401077	push edi		
00401078	push ebx		
00401079	push ecx		
0040107A	push edx		
0040107B	push esi		
0040107C	push edi		
0040107D	push ebx		
0040107E	push ecx		
0040107F	push edx		
00401080	push esi		
00401081	push edi		
00401082	push ebx		
00401083	push ecx		
00401084	push edx		
00401085	push esi		
00401086	push edi		
00401087	push ebx		
00401088	push ecx		
00401089	push edx		
0040108A	push esi		
0040108B	push edi		
0040108C	push ebx		
0040108D	push ecx		
0040108E	push edx		
0040108F	push esi		
00401090	push edi		
00401091	push ebx		
00401092	push ecx		
00401093	push edx		
00401094	push esi		
00401095	push edi		
00401096	push ebx		
00401097	push ecx		
00401098	push edx		
00401099	push esi		
0040109A	push edi		
0040109B	push ebx		
0040109C	push ecx		
0040109D	push edx		
0040109E	push esi		
0040109F	push edi		
004010A0	push ebx		
004010A1	push ecx		
004010A2	push edx		
004010A3	push esi		
004010A4	push edi		
004010A5	push ebx		
004010A6	push ecx		
004010A7	push edx		
004010A8	push esi		
004010A9	push edi		
004010AA	push ebx		
004010AB	push ecx		
004010AC	push edx		
004010AD	push esi		
004010AE	push edi		
004010AF	push ebx		
004010B0	push ecx		
004010B1	push edx		
004010B2	push esi		
004010B3	push edi		
004010B4	push ebx		
004010B5	push ecx		
004010B6	push edx		
004010B7	push esi		
004010B8	push edi		
004010B9	push ebx		
004010BA	push ecx		
004010BB	push edx		
004010BC	push esi		
004010BD	push edi		
004010BE	push ebx		
004010BF	push ecx		
004010C0	push edx		
004010C1	push esi		
004010C2	push edi		
004010C3	push ebx		
004010C4	push ecx		
004010C5	push edx		
004010C6	push esi		
004010C7	push edi		
004010C8	push ebx		
004010C9	push ecx		
004010CA	push edx		
004010CB	push esi		
004010CC	push edi		
004010CD	push ebx		
004010CE	push ecx		
004010CF	push edx		
004010D0	push esi		
004010D1	push edi		
004010D2	push ebx		
004010D3	push ecx		
004010D4	push edx		
004010D5	push esi		
004010D6	push edi		
004010D7	push ebx		
004010D8	push ecx		
004010D9	push edx		
004010DA	push esi		
004010DB	push edi		
004010DC	push ebx		
004010DD	push ecx		
004010DE	push edx		
004010DF	push esi		
004010E0	push edi		
004010E1	push ebx		
004010E2	push ecx		
004010E3	push edx		
004010E4	push esi		
004010E5	push edi		
004010E6	push ebx		
004010E7	push ecx		
004010E8	push edx		
004010E9	push esi		
004010EA	push edi		
004010EB	push ebx		
004010EC	push ecx		
004010ED	push edx		
004010EE	push esi		
004010EF	push edi		
004010F0	push ebx		
004010F1	push ecx		
004010F2	push edx		
004010F3	push esi		
004010F4	push edi		
004010F5	push ebx		
004010F6	push ecx		
004010F7	push edx		
004010F8	push esi		
004010F9	push edi		
004010FA	push ebx		
004010FB	push ecx		
004010FC	push edx		
004010FD	push esi		
004010FE	push edi		
004010FF	push ebx		

Project page: <https://bitbucket.org/mihaila/bindead>