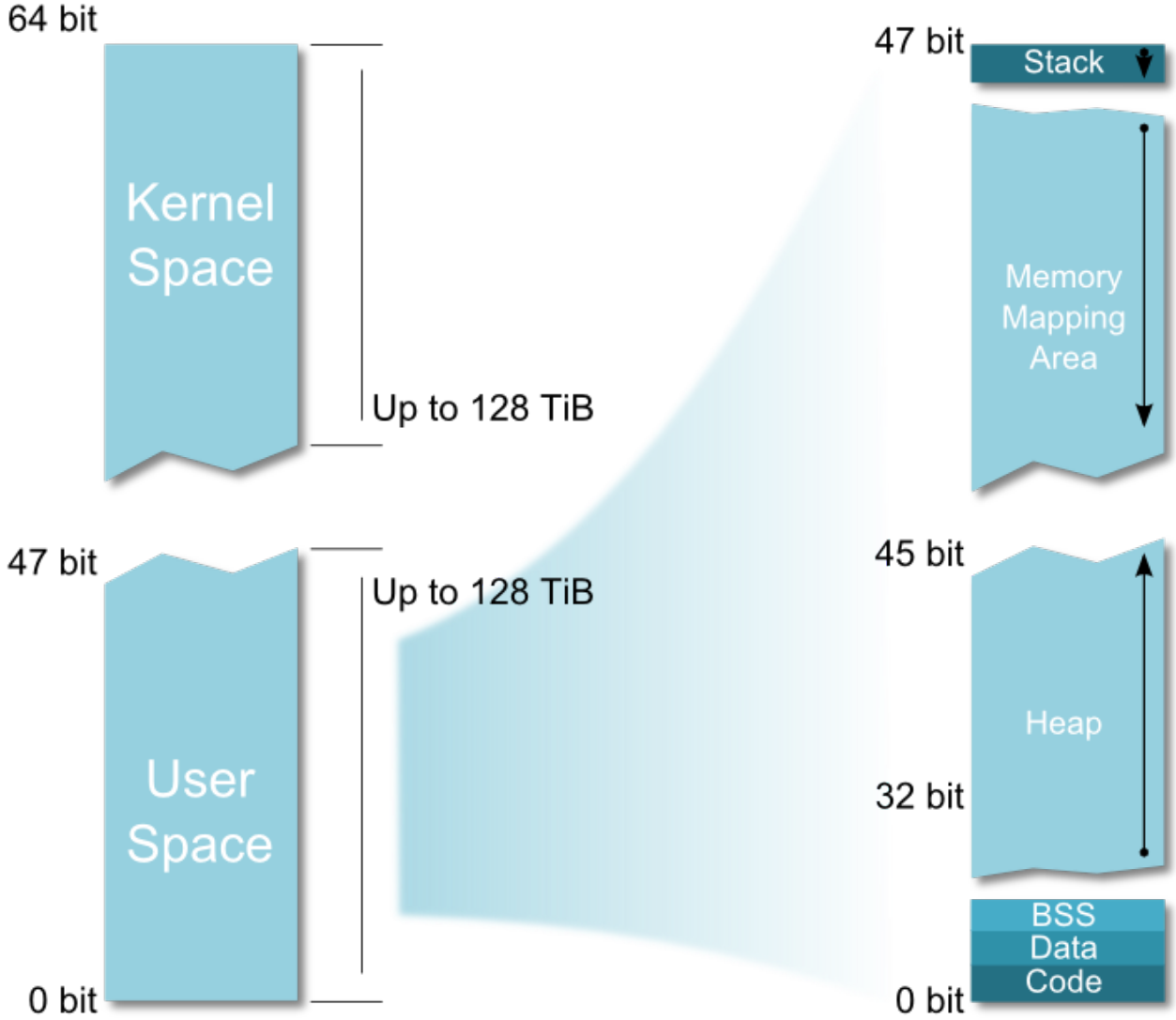
Department of Computer Science

UNIVERSITY OF OXFORD

Automated Black Hats

Application Memory



https://www.berthon.eu/wiki/foss:wikishelf:linux:memory

Application Memory

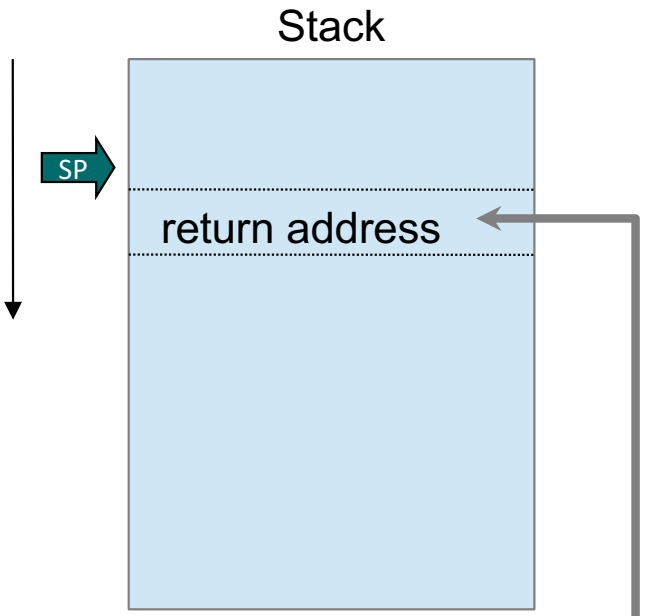
```
int global;


int main()
{
  int local;
  printf("global: %16p\n", &global);
  printf("main:   %16p\n", &main);
  printf("local:  %16p\n", &local);
  printf("heap:   %16p\n", malloc(4));
}
```

```
global:       0x108f41020
main:         0x108f40ed0
local:     0x7fff56cbf8dc
heap:      0x7ff5814026d0
```

What is the Stack used for?

- The stack enables functions/procedures
- Stores
  - Procedure-local data
  - Return address of calls
- Benefits:
  - More economical
  - Permits recursion

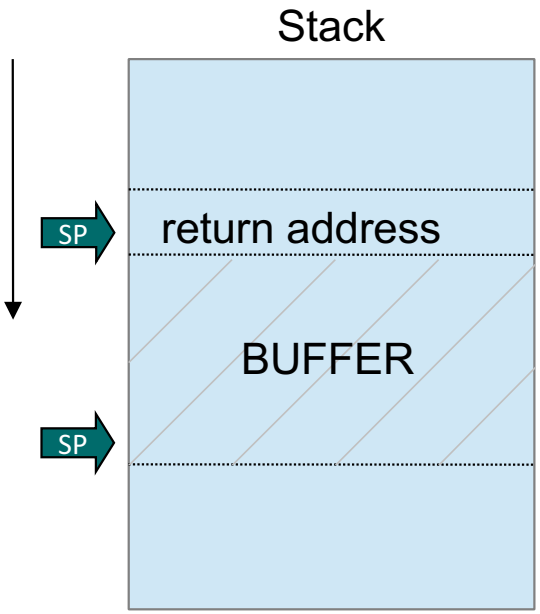



```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}


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

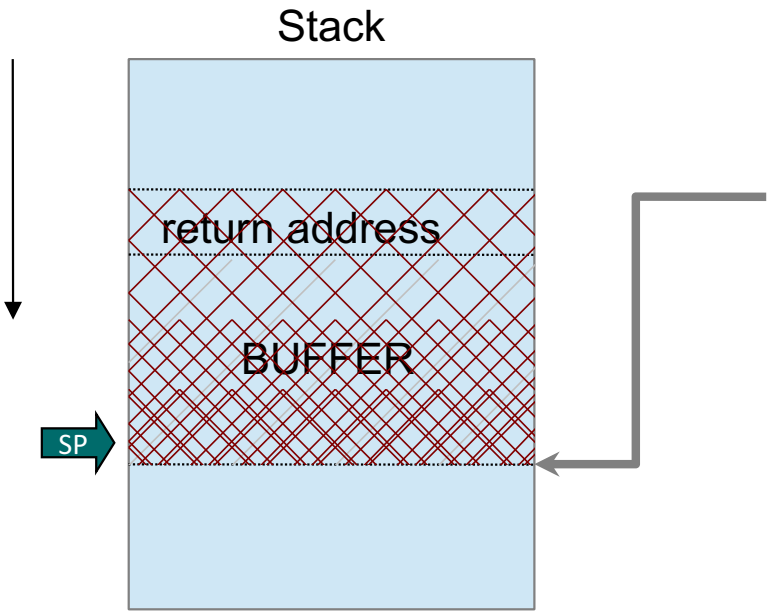
IP




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

        …
        strcpy(buffer, INPUT);
        …
    }


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

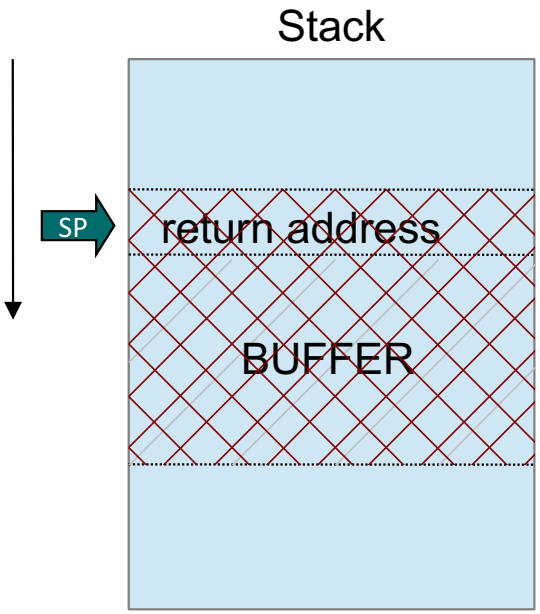



```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}

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

IP

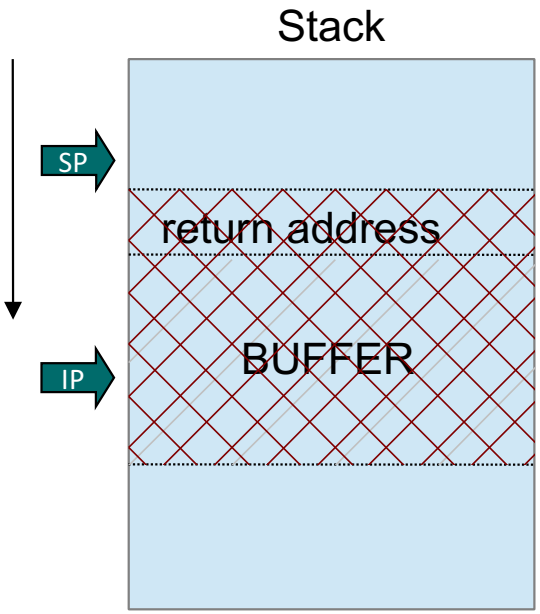



```
void f(void)
{
    char buffer[100];

    …
    strcpy(buffer, INPUT);
    …
}

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

IP




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

    …
    strcpy(buffer, INPUT);
    …
}


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

Other Options

- C++ gives rise to a large number of *function pointers* to implement virtual methods
- These are typically on the heap
- Exploit like return addresses
- There are similar pointers for dynamically loaded libraries

Countermeasures

- Address space layout randomization (ASLR)
- Execution protection (NX) for stack and heap
- Read-only code segments (W^X)
- Stack canaries
- Control-flow integrity

Countermeasure: ROP

- “Return-oriented programming”
- Rummage through existing code (say libc) to find return instructions (0xc3)
- Any instructions before these can do useful work
- Place address of next instruction on stack
- Countermeasure: ASCII armoring

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

Hacking with BMC

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

Hacking with BMC

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

- 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 **unroll the loop twice** and accelerate the result

```
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 www.github.com/diffblue/cbmc

Making this Real

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

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

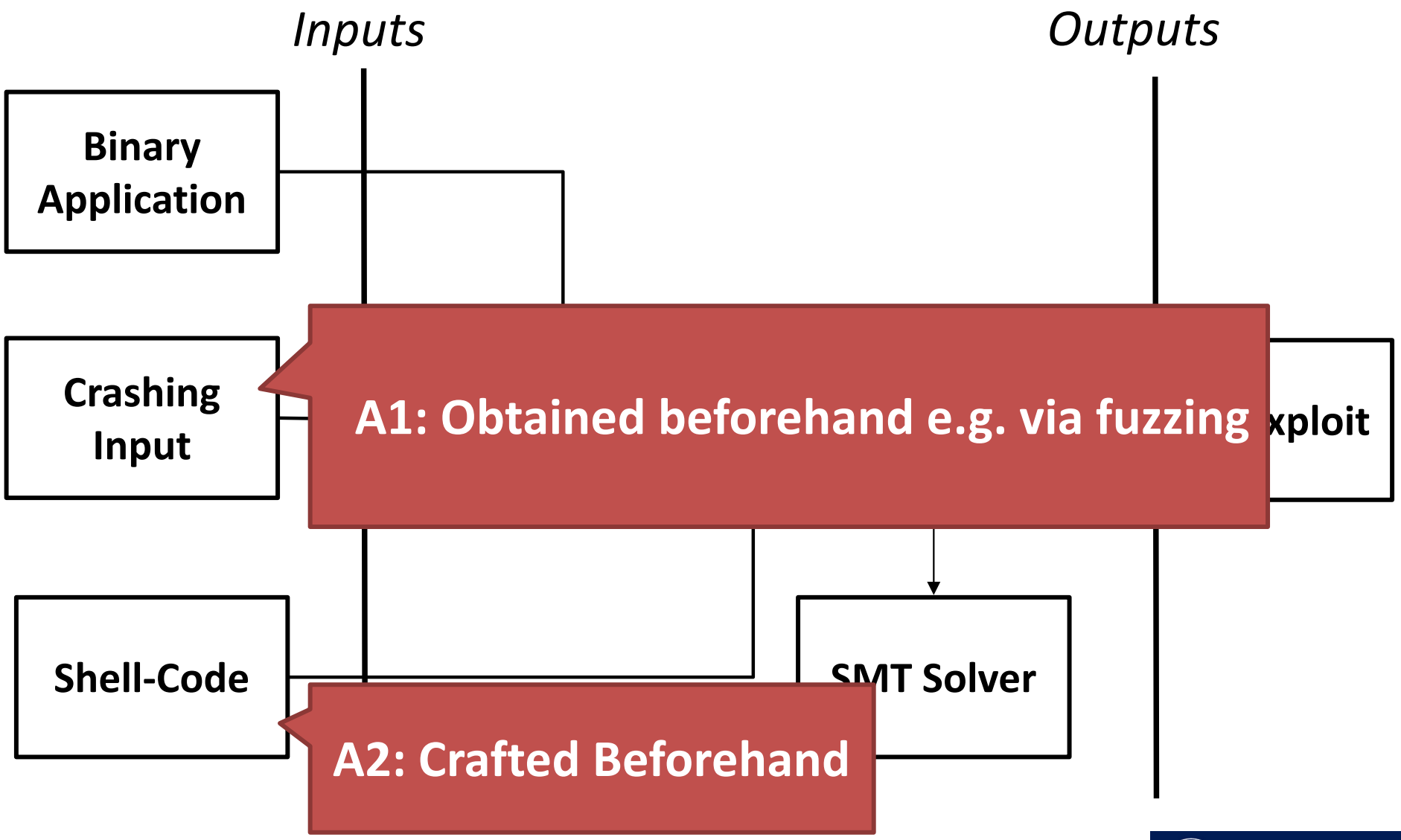
The Hemiptera Bug Suite

Project	# Invalid Accesses	# Failed Asserts	# Div By Zero	# Dang Ptr. Use	Avg. Lines of C Code
JasPer	11	6	2	0	25,891
LibTIFF	36	1	4	0	56,573
tcpdump	10	0	0	0	47,782
Libjpeg-turbo	1	0	1	0	25,920
zlib	2	0	0	1	7,835
file	5	1	1	0	12,597
w3m	44	0	0	2	53,338
FLAC	5	0	0	0	46,114
TOTAL	**114**	**8**	**8**	**3**	**276,050**

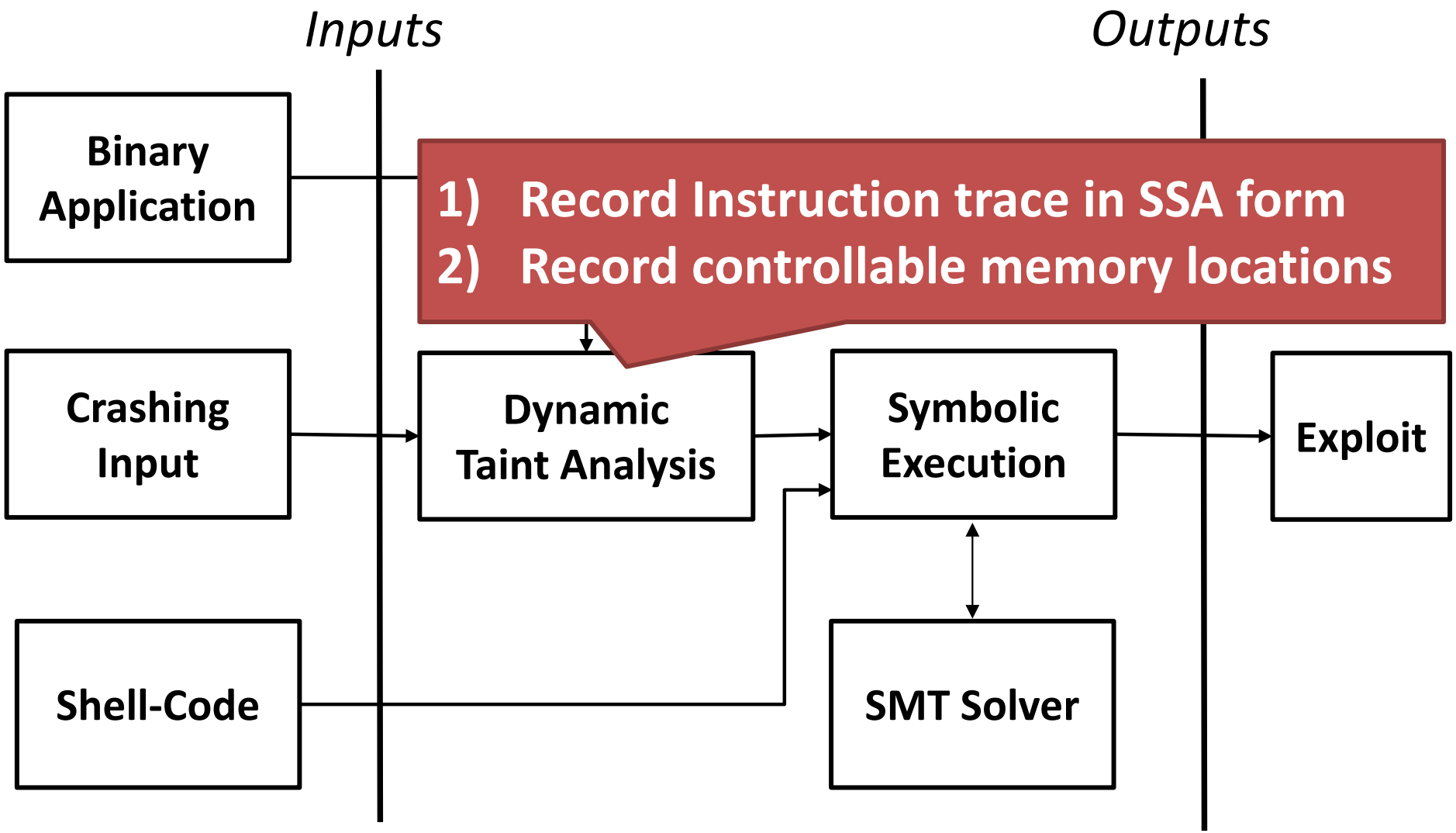
Analysis of Binaries using Taint

- Adobe Acrobat executes ca. 1bn instructions before dealing with the PDF
- Just too much even for the best solvers
- Idea: build partial formula, focussing on parts that are influenced by input

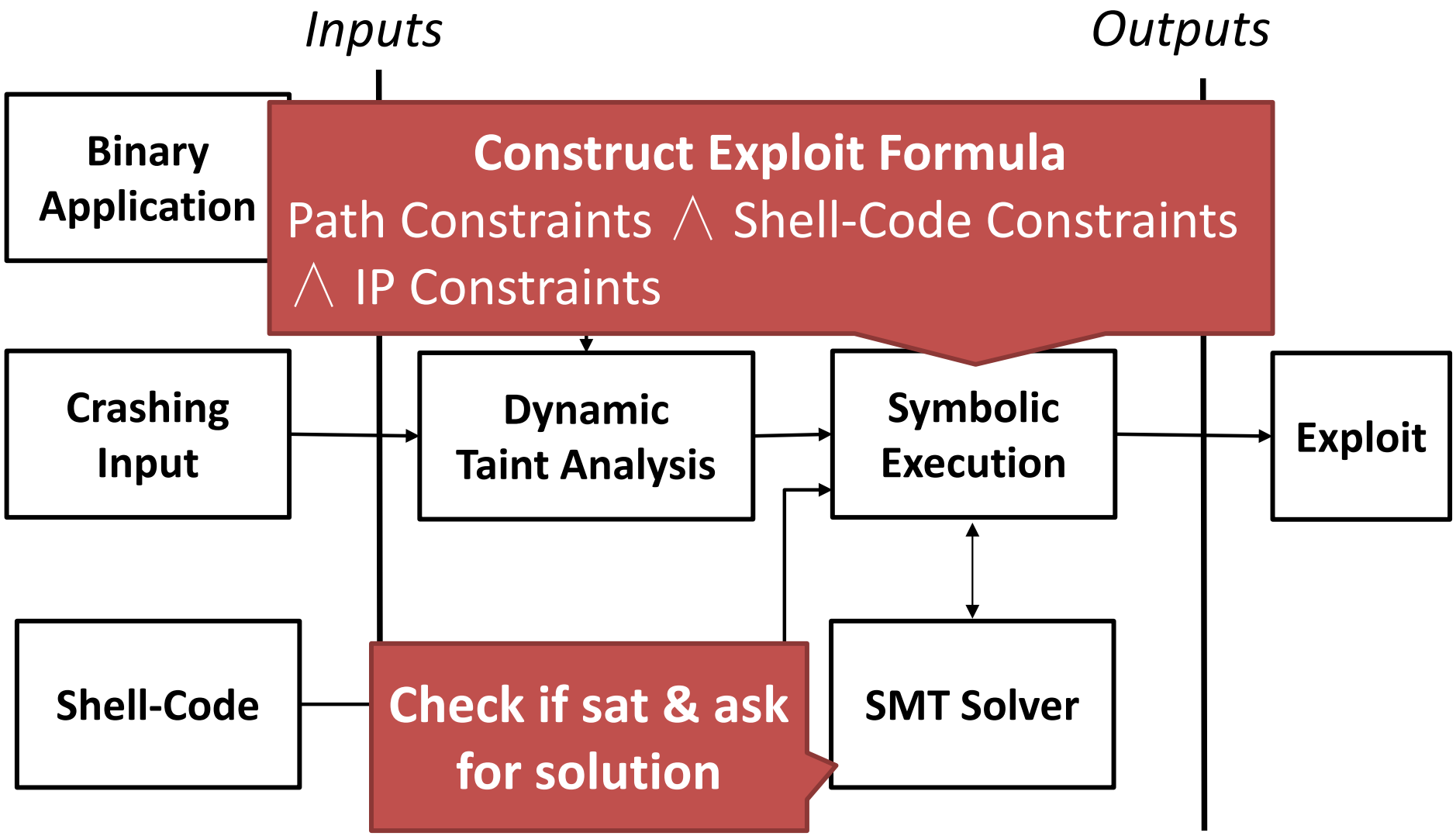
Overview: Taint Analysis



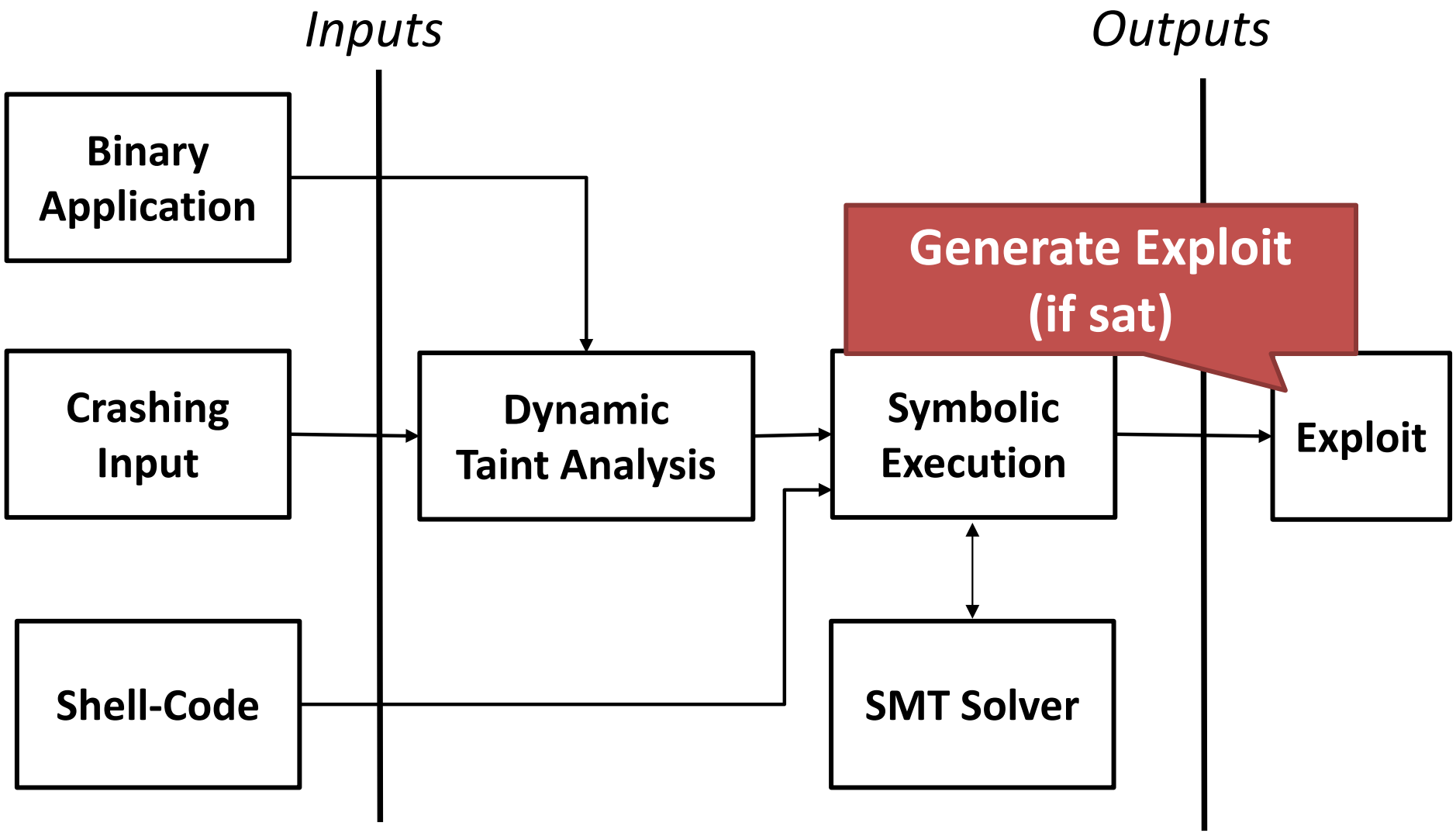
Overview: Taint Analysis



Overview: Taint Analysis



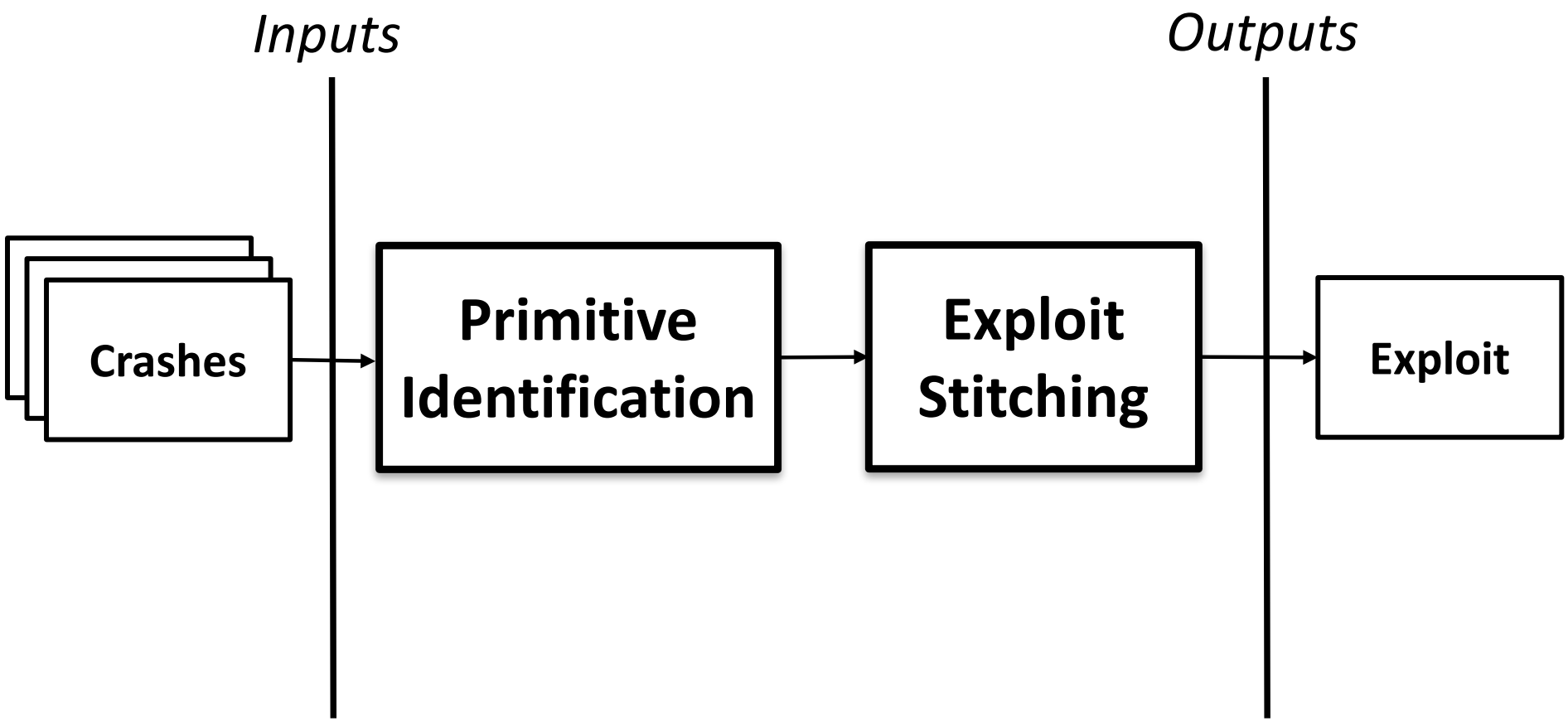
Overview: Taint Analysis



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 eyeing up Debian...)

Goal: Exploit Stitching



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