Resilient Software

PROFESSIONAL EDUCATION

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Automatically Discovering and Eliminating Software Defects

Martin Rinard

MIT EECS, MIT CSAIL Professor

Computer Science and Artificial Intelligence Laboratory (CSAIL)

Massachusetts Institute of Technology

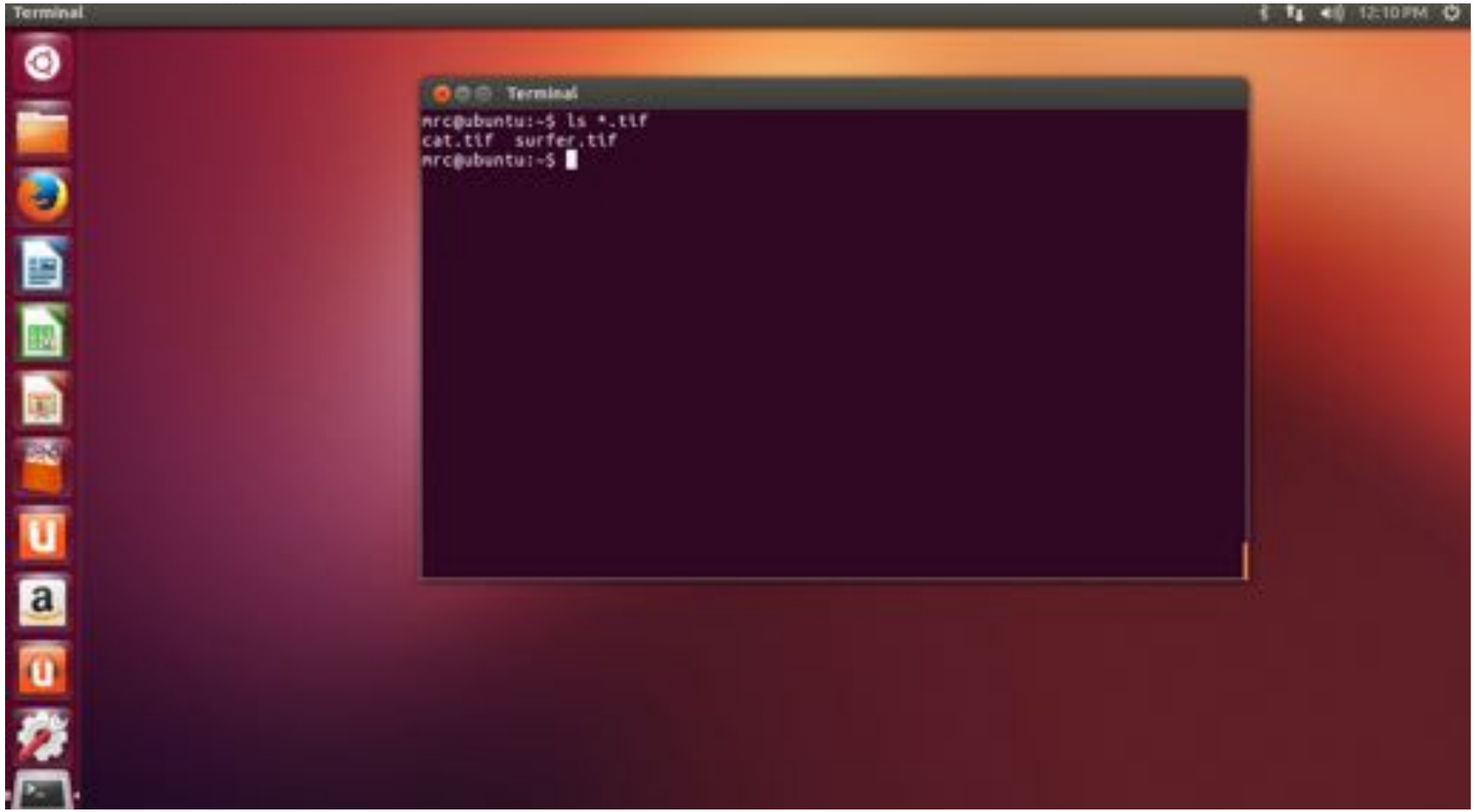
PROFESSIONAL EDUCATION

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

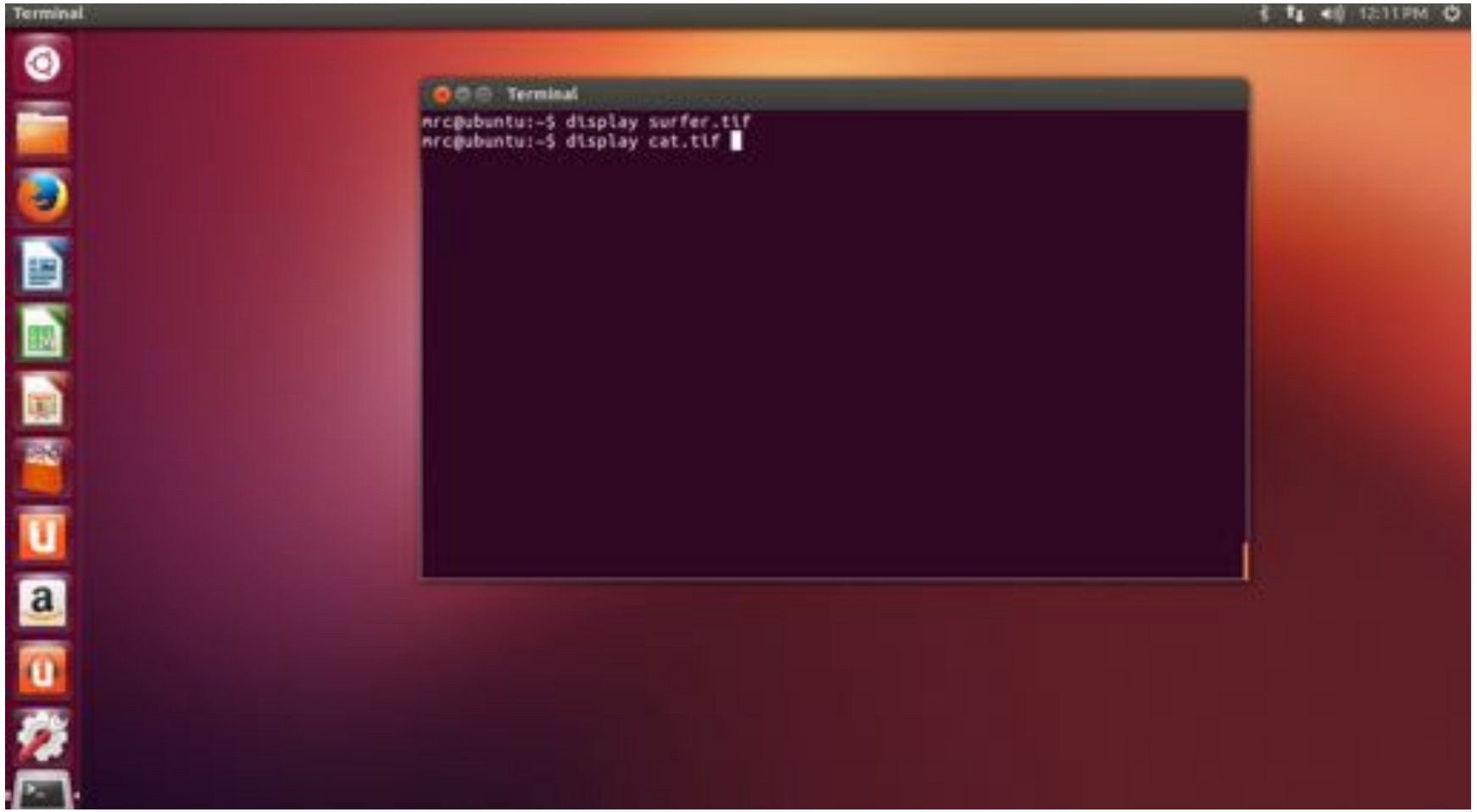
ImageMagick Display

- Popular, free and open-source software suite
  - Displaying, converting and editing images
  - Read/Write > 200 formats
- Very popular for users && programs
  - Drupal, MediaWiki, phpBB, Vbulletin, etc.,

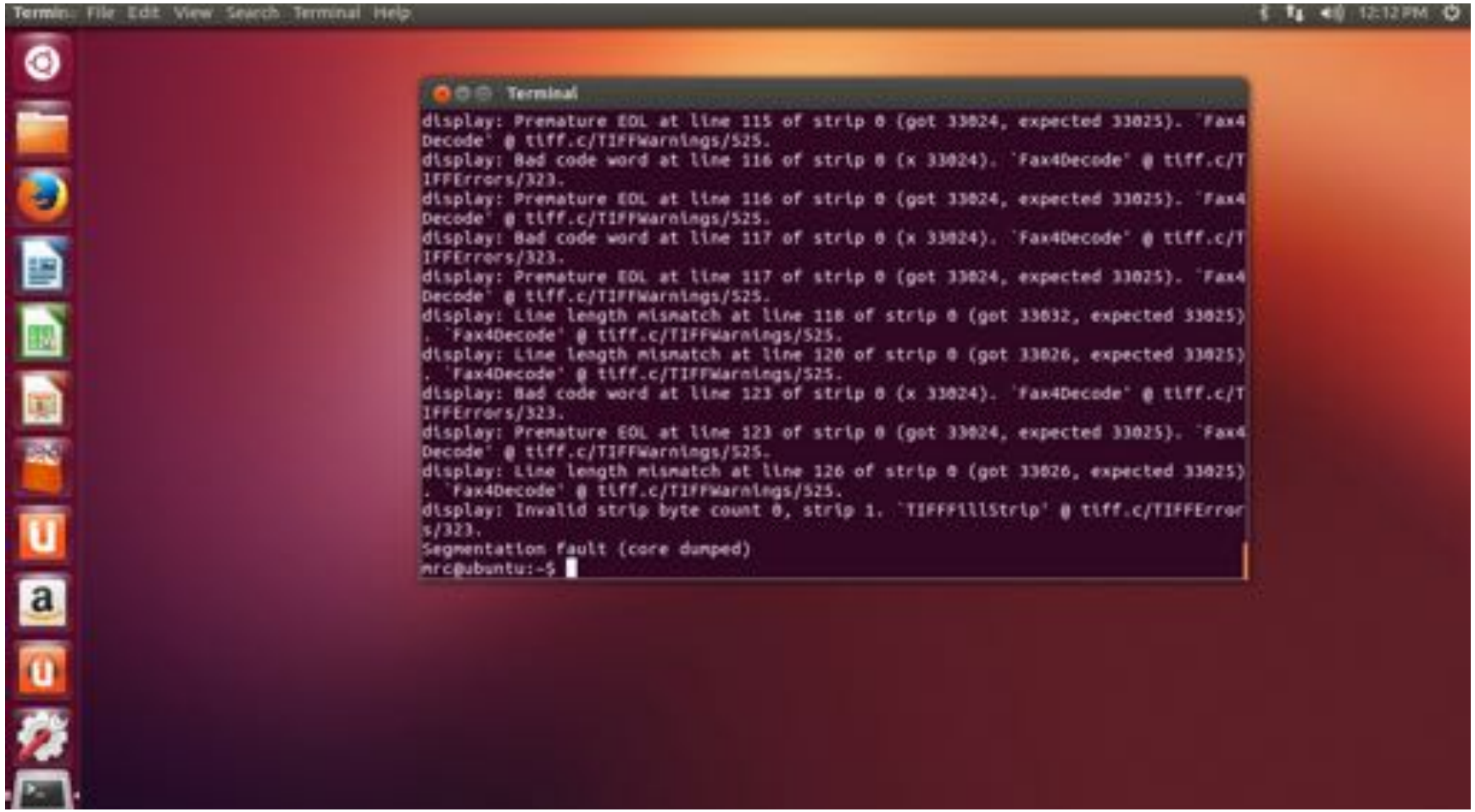
Images



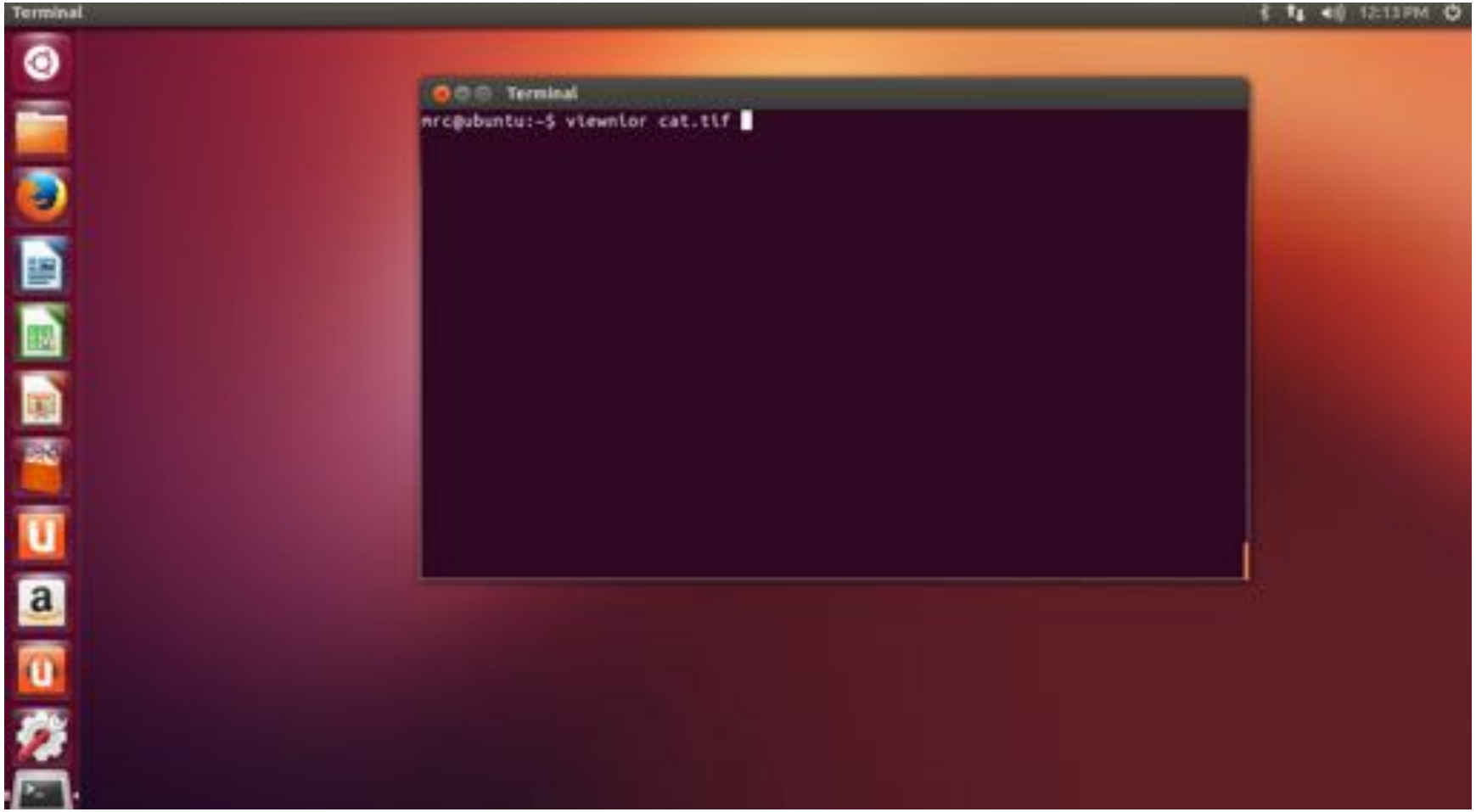
Display Cat



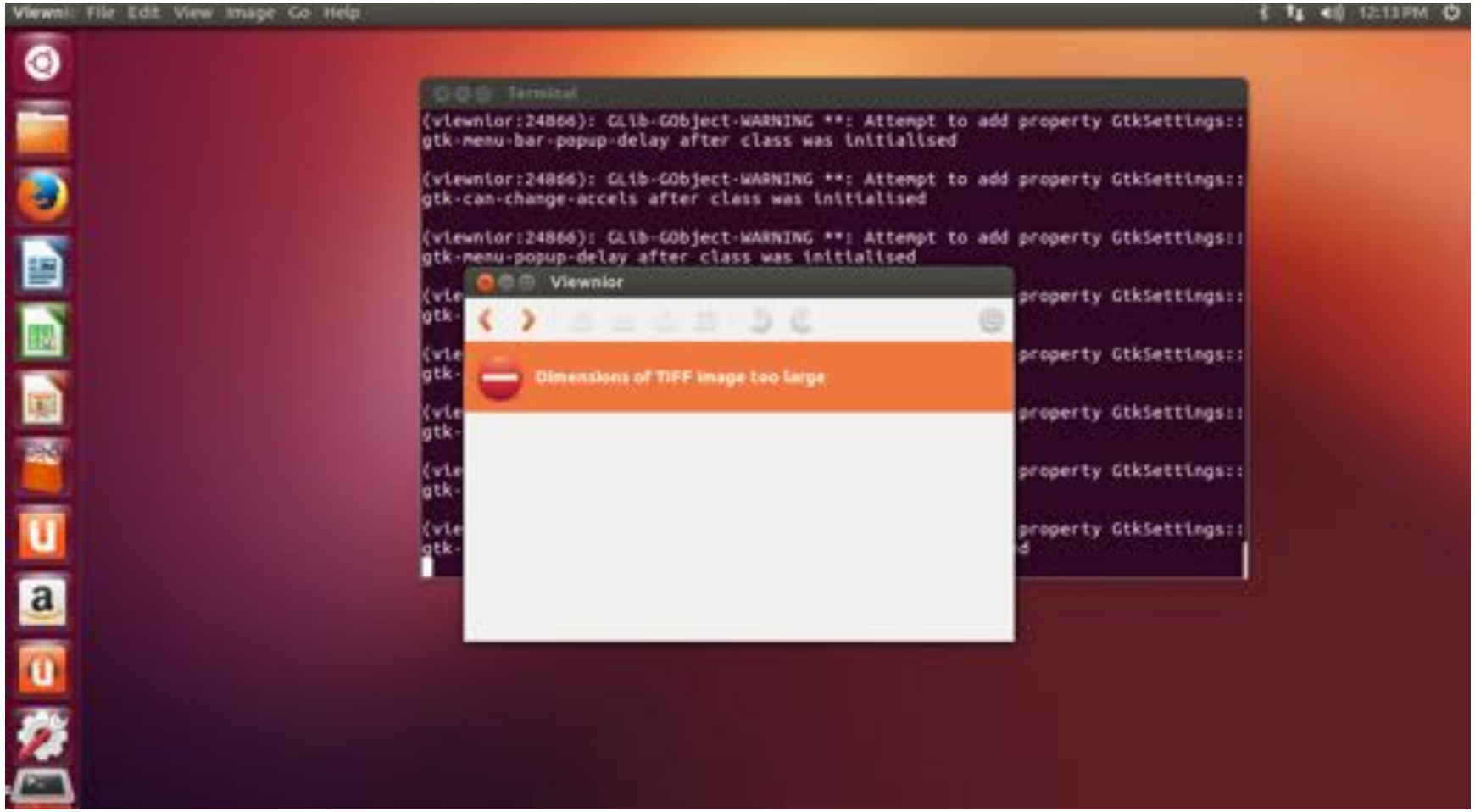
Cat Crash



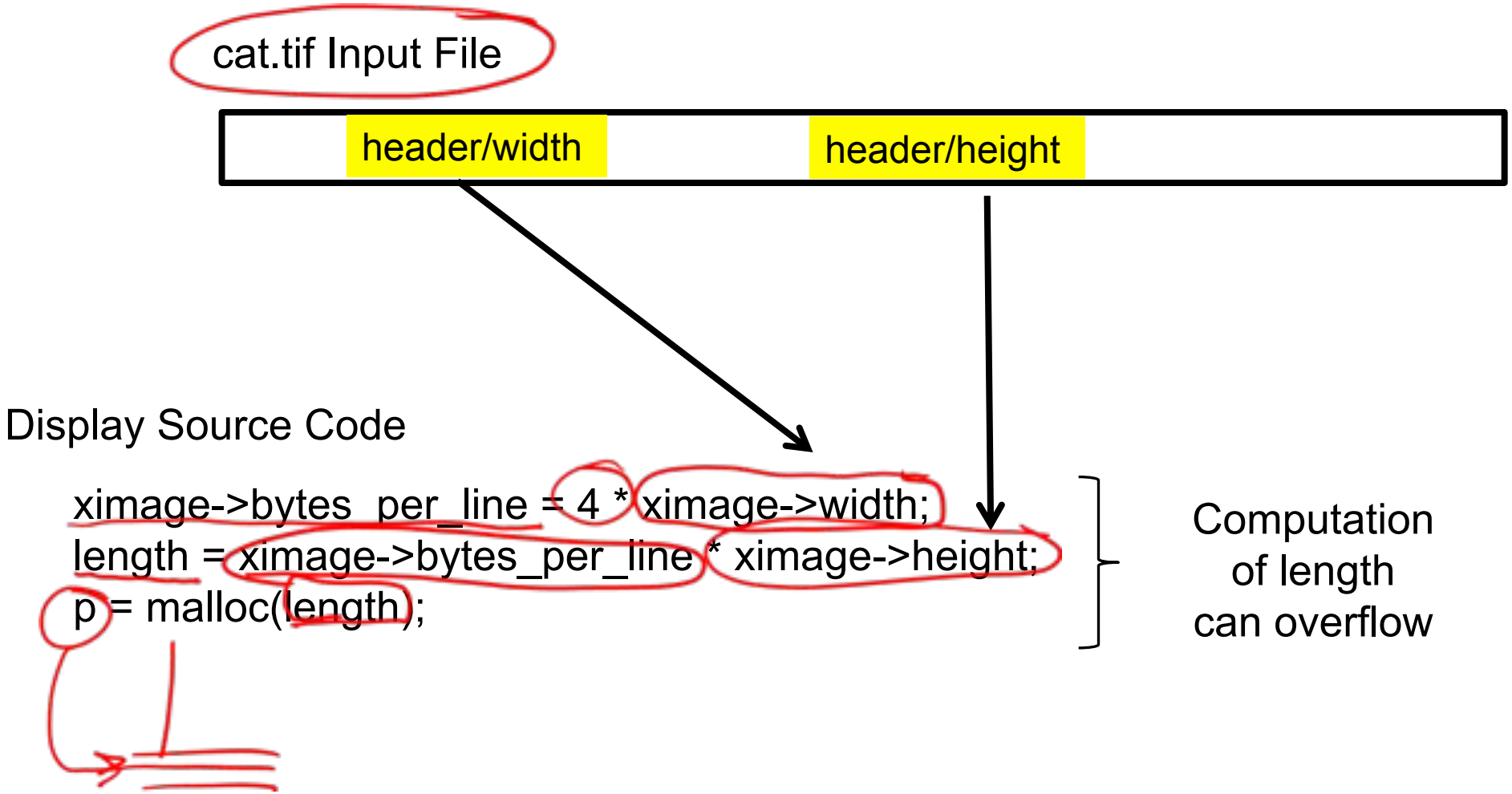
ViewNior Cat



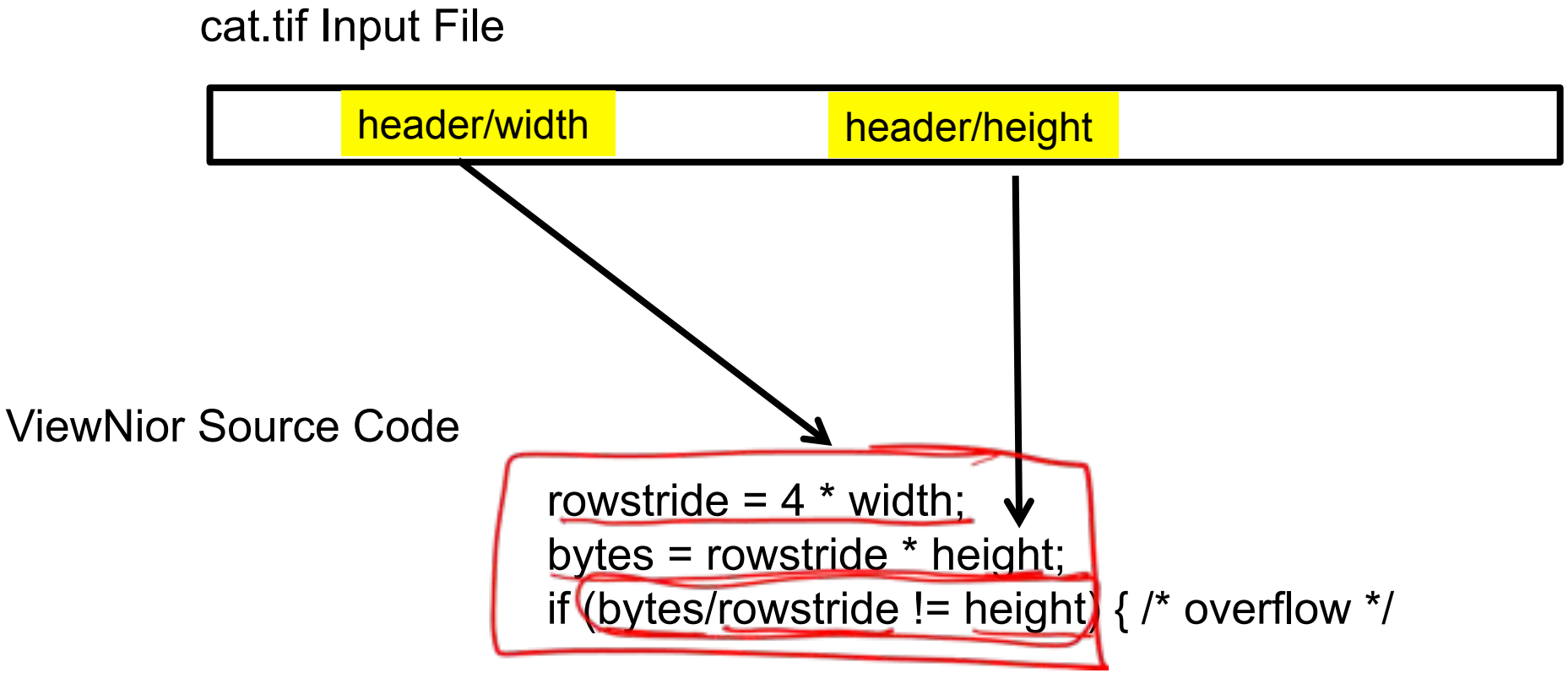
ViewNior Check



Missing Integer Overflow Check in Display



(Risky) Integer Overflow Check in ViewNior



How To Check For Integer Overflow

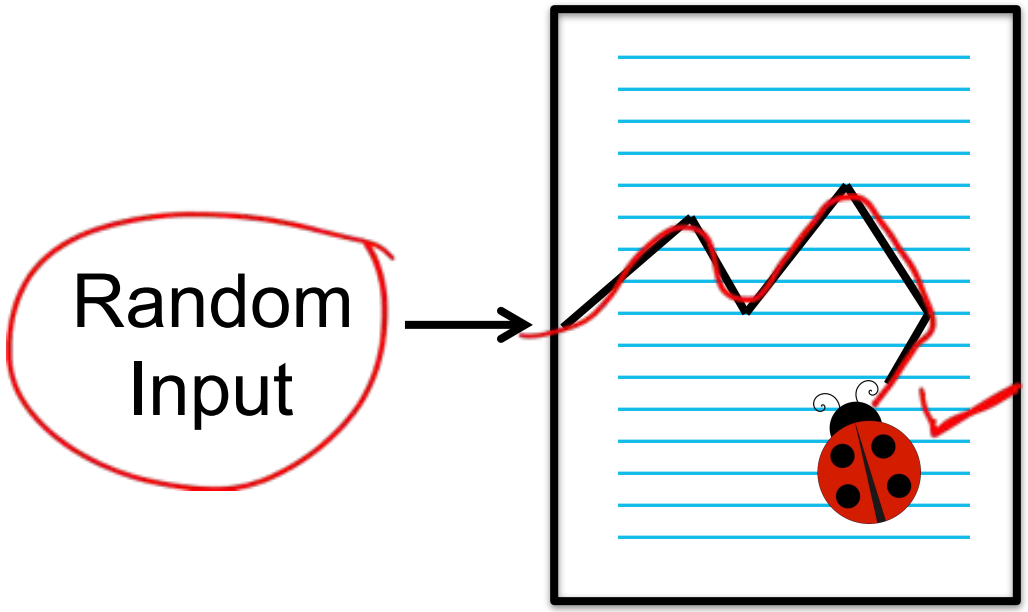
```
uint32_t i, j;
if ((uint64_t)i * (uint64_t)j > MAX_UINT32) {
  /* overflow */
}

int32_t k, l;
if (((int64_t)k * (int64_t)l < MIN_INT32) ||
    ((int64_t)k * (int64_t)l > MAX_INT32)) {
  /* overflow */
}
```

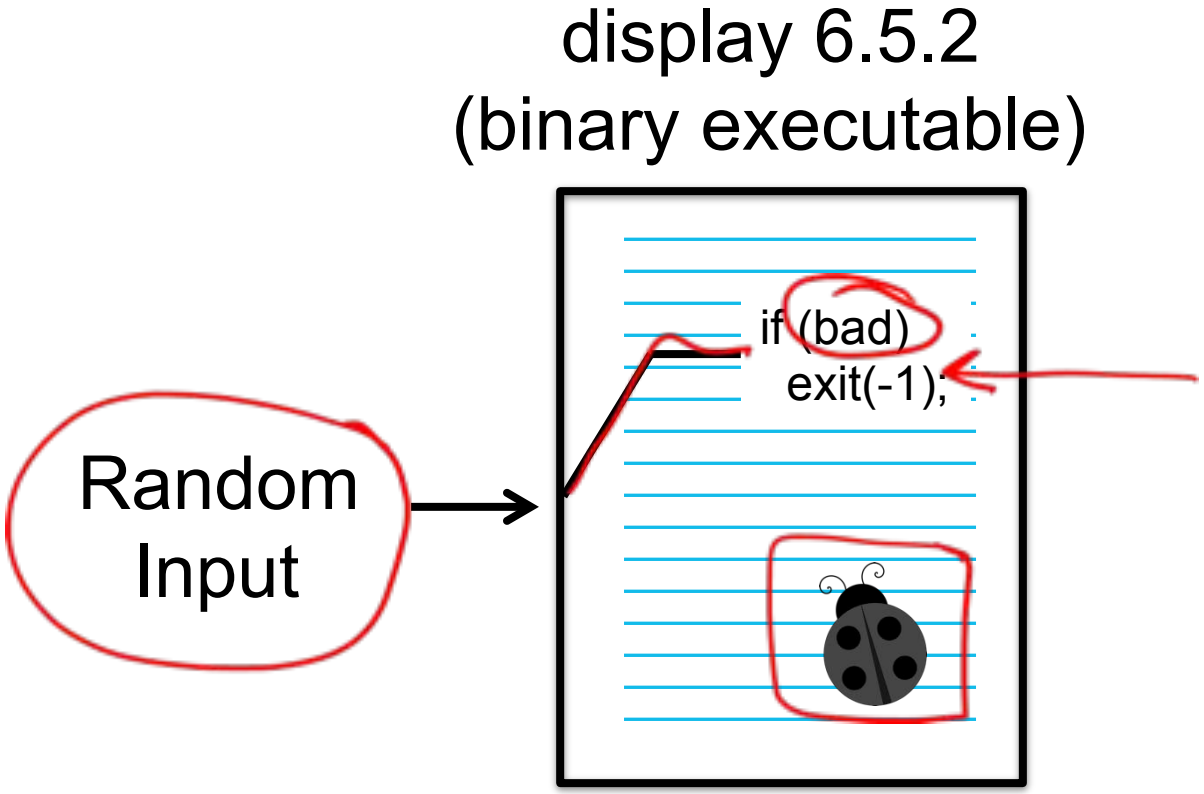
How To Find Integer Overflow Errors

Fuzz Testing (Ideal)

display 6.5.2
(binary executable)

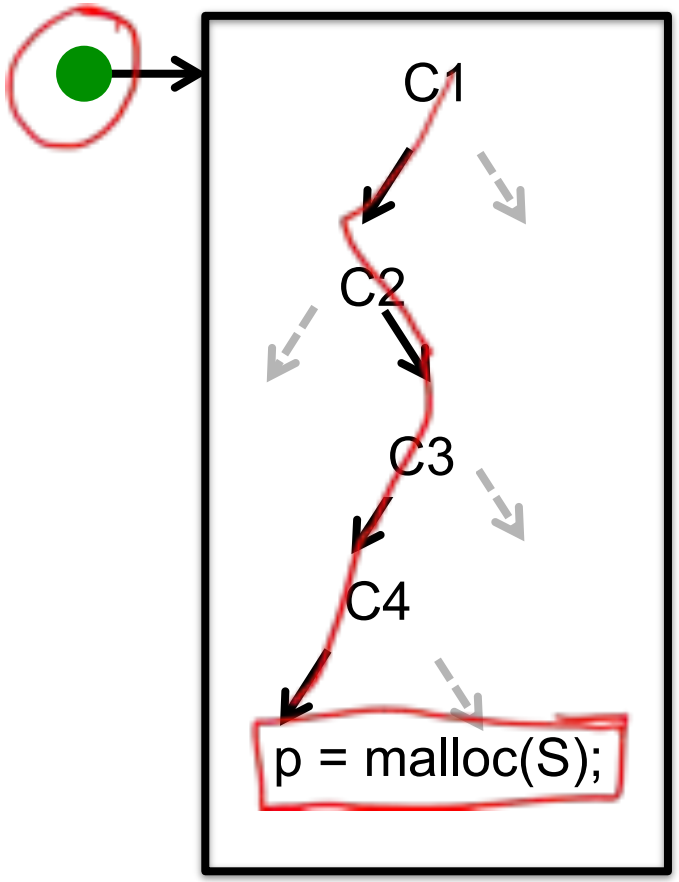


Fuzz Testing (Usually In Practice)



Finding Integer Overflows With DIODE

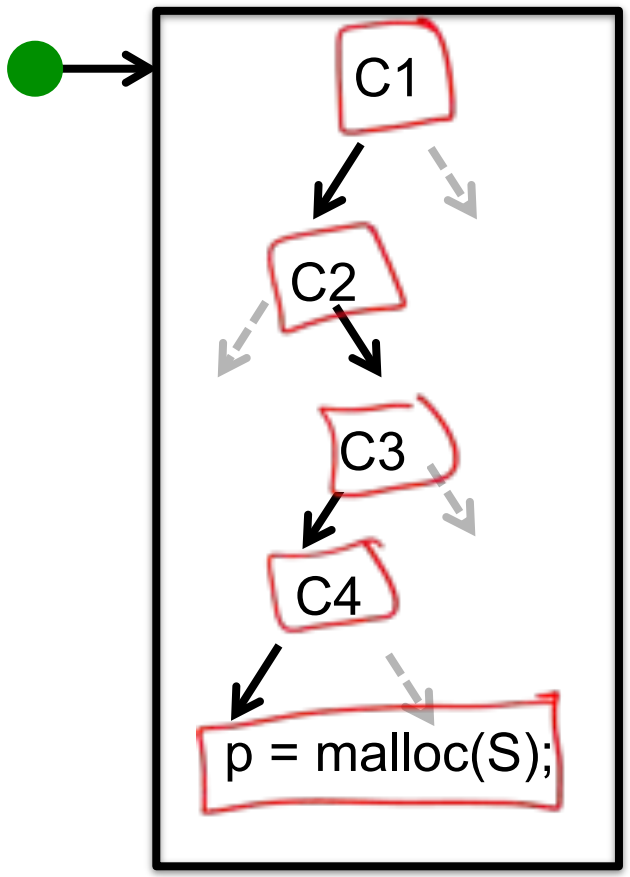
display 6.5.2



- Run (instrumented) application on benign input
- Encounter malloc(S) site
- Use solver to find new input that overflows S
- Run application on new input, see what happens
  - If reaches malloc(S) site
  - Guaranteed overflow!

Common Outcome

display 6.5.2



display 6.5.2

Why

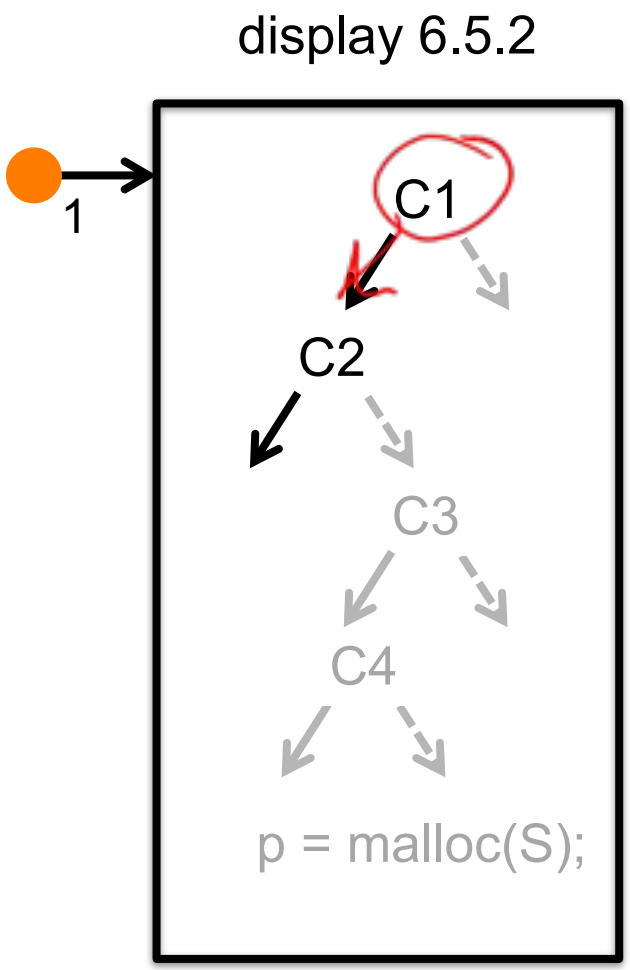
- Sanity checks

Next Try

Solver produces new input that
1. Overflows S, and
2. Follows same path as benign input

Common outcome
UNSAT - no such new input exists

What DIODE Uses (Conceptually)



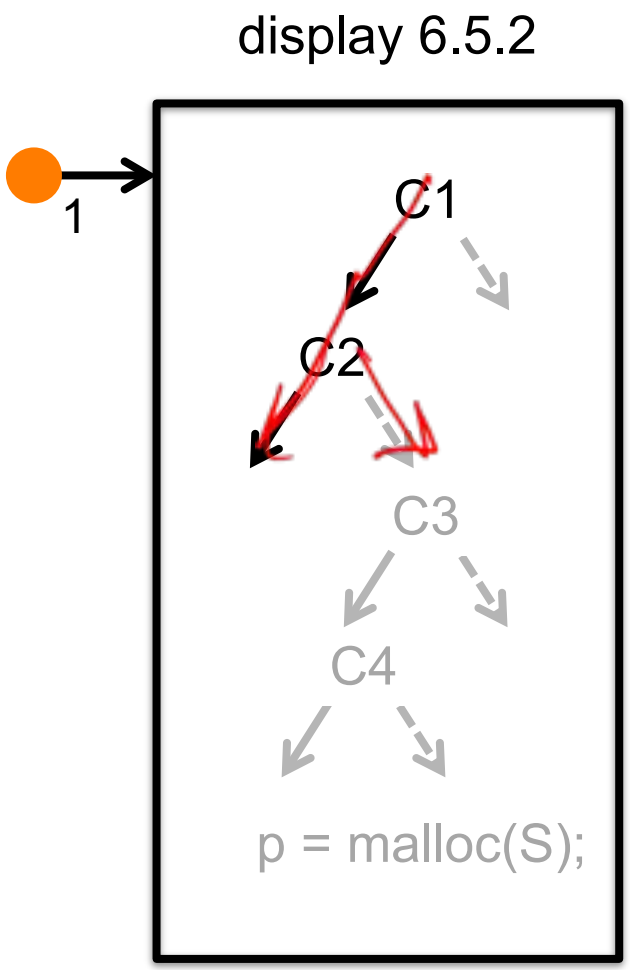
Top Down Branch Enforcement

- Solver finds new input $\bullet_1$ that

1. Satisfies C1 and
2. Overflows S

If $\bullet_1$ reaches malloc(S), integer overflow!

What DIODE Uses (Conceptually)



Top Down Branch Enforcement

- Solver finds new input $\bullet_1$ that

1. Satisfies C1 and
2. Overflows S

If $\bullet_1$ reaches malloc(S), integer overflow!

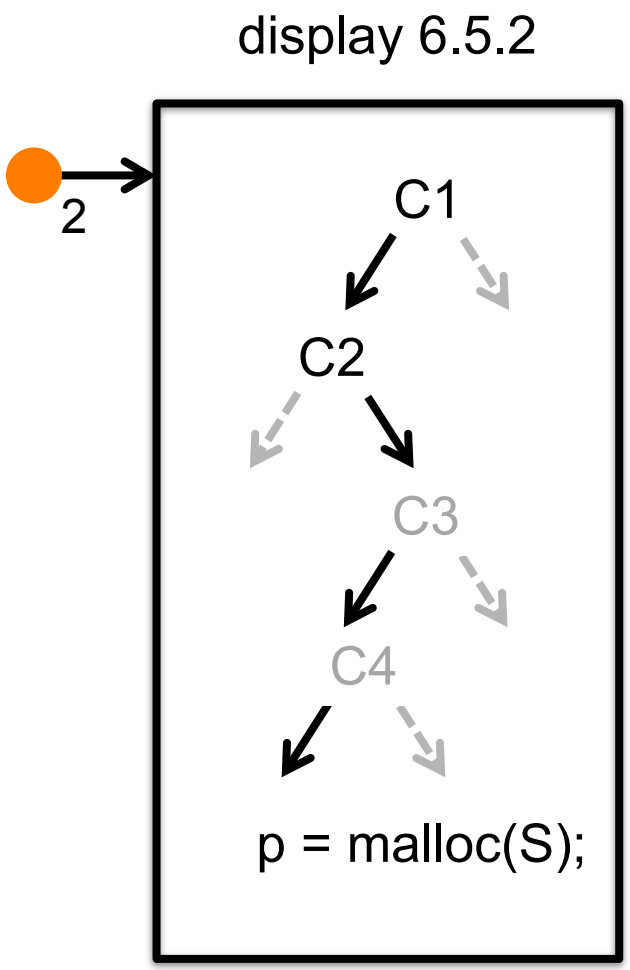
Otherwise solver finds new input $\bullet_2$ that

1. Satisfies C1,
2. Satisfies not C2, and
3. Overflows S

If $\bullet_2$ reaches malloc(S), integer overflow!

…

What DIODE Uses (Conceptually)



Top Down Branch Enforcement

- Solver finds new input $\bullet_1$ that

1. Satisfies C1 and
2. Overflows S

If $\bullet_1$ reaches malloc(S), integer overflow!

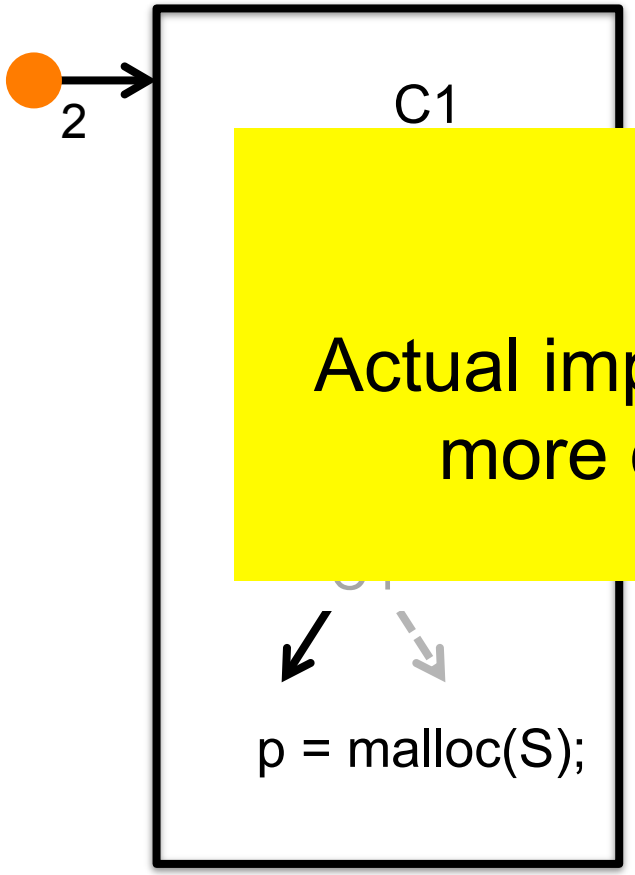
Otherwise solver finds new input $\bullet_2$ that

1. Satisfies C1,
2. Satisfies not C2, and
3. Overflows S

If $\bullet_2$ reaches malloc(S), integer overflow!

…

What DIODE Uses (Conceptually)



Top Down Branch Enforcement

- Solver finds new input $\bullet_1$ that
  1. Satisfies C1 and
  2. Satisfies not C2, and
  3. Overflows S

overflow!

$\bullet_2$ that

If $\bullet_2$ reaches malloc(S), integer overflow!

…

Warning

Actual implementation is significantly more complicated than this…

DIODE Integer Overflow Results

Application	Format	Error Location	Defect Detection Time	Enforced Branches	Error Source
cwebp 0.3.1	jpeg	jpegdec.c:248	10 sec	0	New
dillo 2.1	png	png.c:203	11 min	3	CVE-2009-2294
dillo 2.1	png	fltkimagebuf.cc:39	15 min	3	New
display 6.5.2	tiff	xwindow.c:5619	10 sec	0	CVE-2009-1882
display 6.5.2	tiff	cache.c:3717	10 sec	0	New
swfplay 0.5.5	swf	jpeg_rgb_decoder.c:192	10 sec	0	New
swfplay 0.5.5	swf	jpeg.c:192	48 min	5	BuzzFuzz

7 Integer Overflow Errors
4 Previously Unknown
3 Require Enforced Branches

Finding Buffer Overflow Errors

Buffer Overflow in gif2tiff

```
unsigned int prefix[4096];
int datasize;
int clear;

readraster(void) {
  register int code;
  datasize = getc(infile);
  clear = 1 << datasize;
  for (code = 0; code < clear; code++) {
    prefix[code] = 0;
  }
}
```

data from gif input

data controls clear

overflow if clear > 4095

Buffer Overflow in gif2tiff

```
unsigned int prefix[4096];

datasize = getc(infile);
clear = 1 << datasize;
for (code = 0; code < clear; code++)
    prefix[code] = 0;
```

size of buffer does not depend on input file

data from input file controls value

same variable

control dependence

Generated constraint system:

code < (1 << byte 415) and code >= 4096

Solution: byte 415 = 30

jasper Buffer Overflow Error

jasper converts jpeg2K files

jpeg2K images can have multiple tiles

jasper has off by one error in tile handling code (checks for >, not >=)

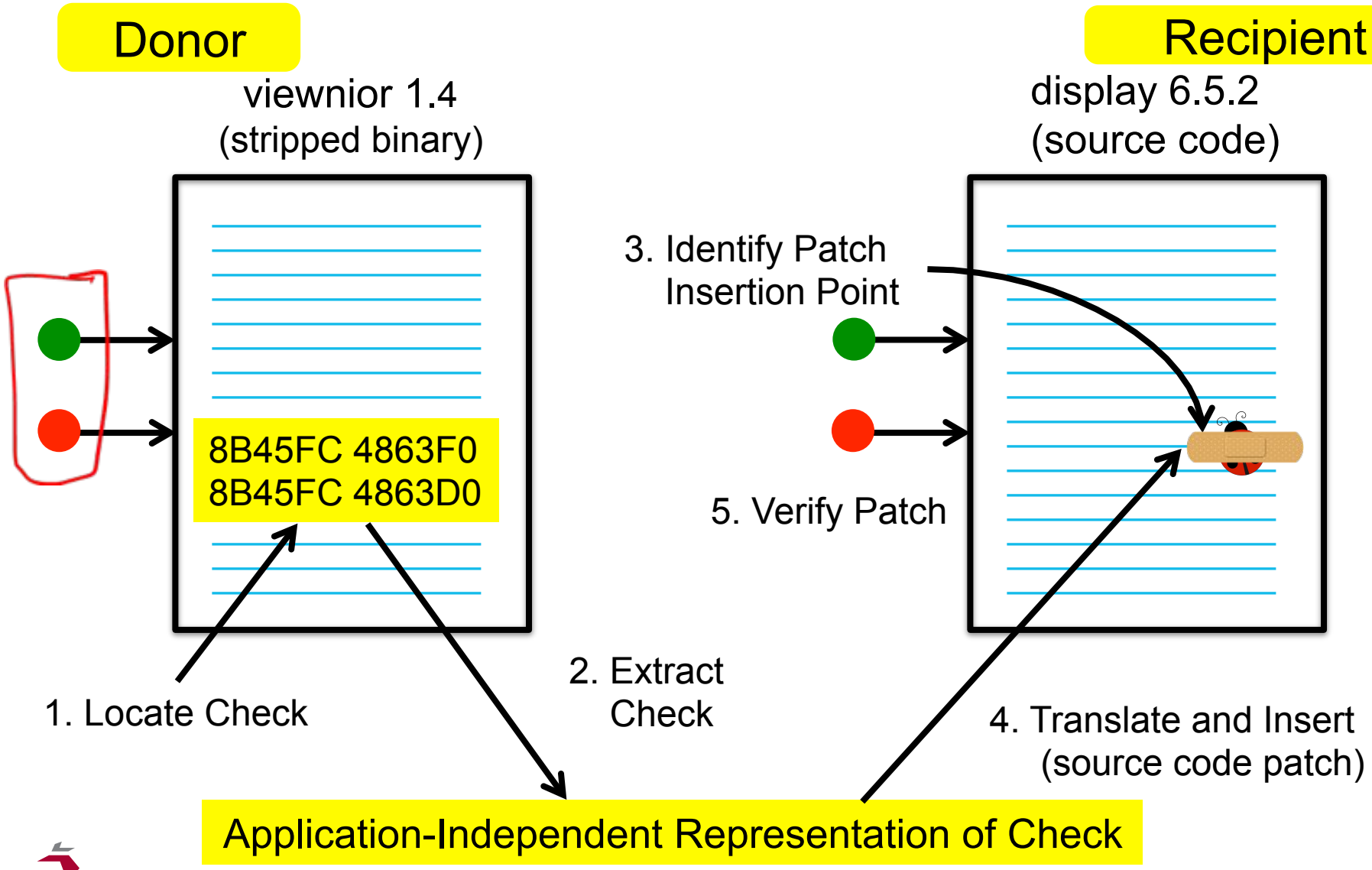
Off by one check (should be >=)

```
if (JAS_CAST(int, sot->tileno) > dec->numtiles) {
        jas_eprintf("invalid tile number in SOT marker segment\n");
        return -1;
}
/* Set the current tile. */
dec->curtile = &dec->tiles[sot->tileno];
```

Out of bounds addr

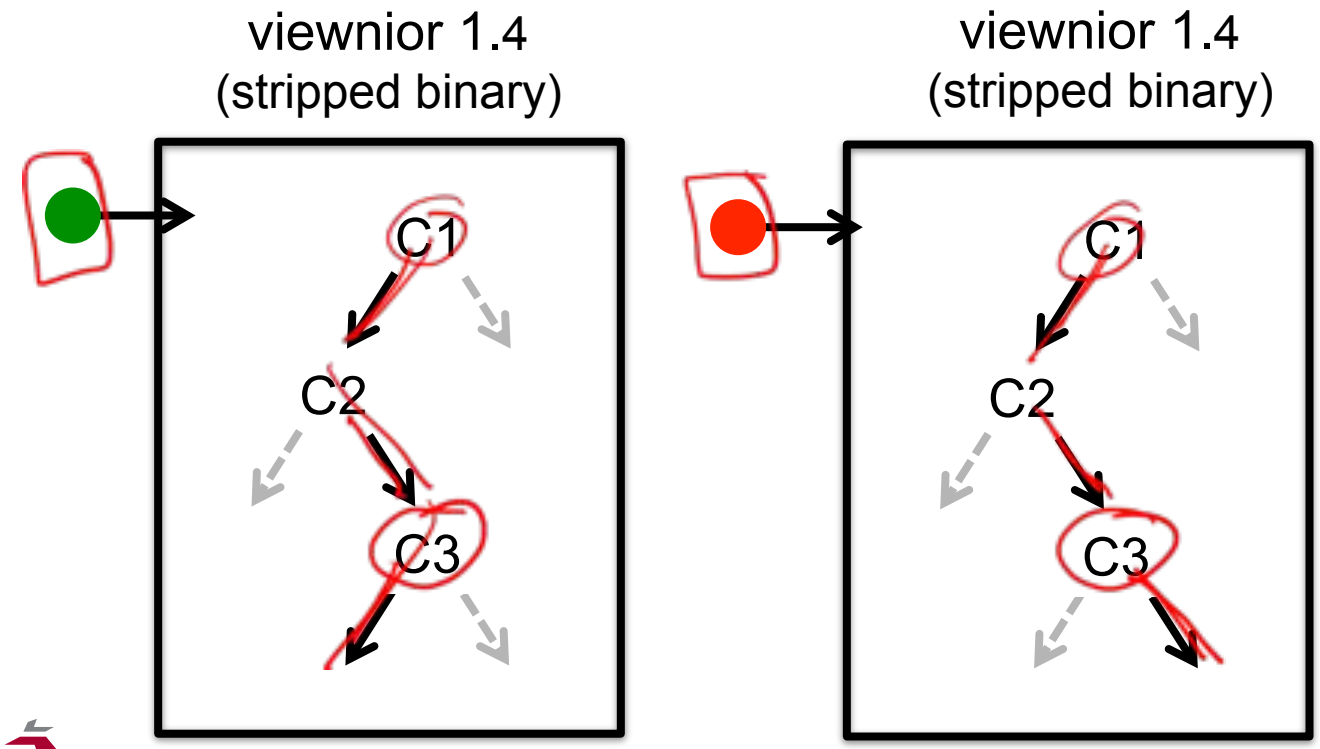
Automatically Patching Errors

CodePhage (CP) Overview



Locate Check

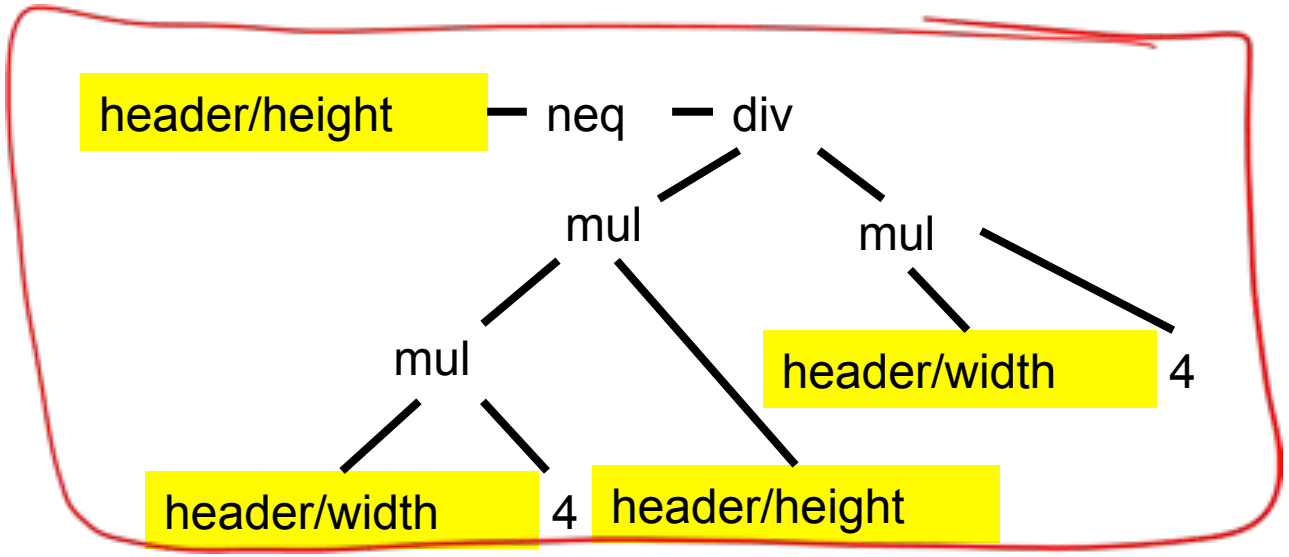
- Execute instrumented version of donor (viewnior 1.4)
- Record trace of executed conditional branches
- Find flipped branches on malicious input



Hypothesis: C3 is the missing check in recipient

Extract Check

- Start with a conditional branch instruction
  jne label, je label, jle label, …
- Goal: Obtain application-independent check
- Symbolic expression tree for condition
  - Internal nodes are operations (add, sub, cmp, …)
  - Leaves are constants and input fields



Integer Overflow Check

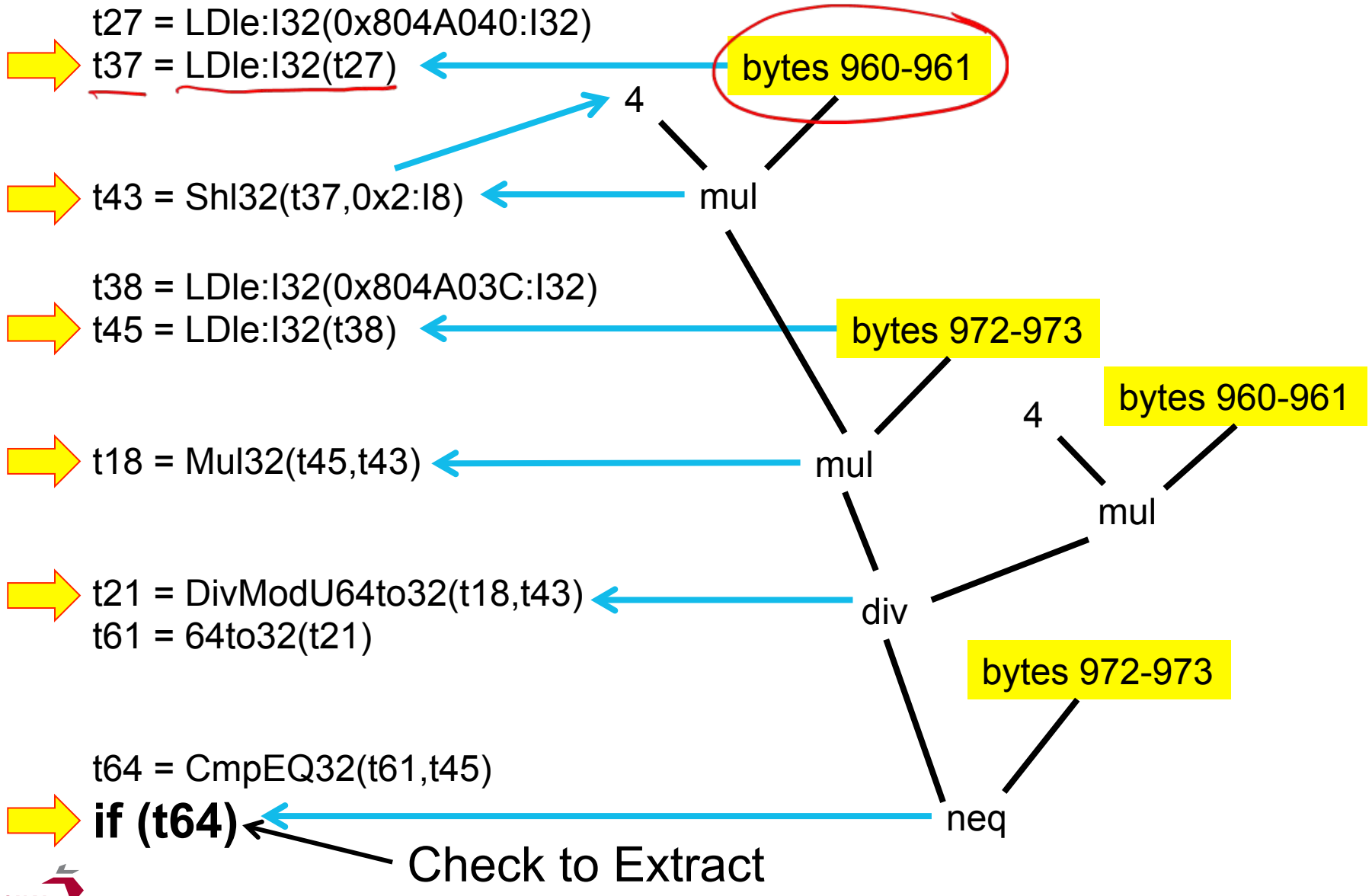
Challenges

- Condition can be computed by arbitrarily complex sequence of binary instructions
  - Need to extract computed logical expression
  - Can involve arbitrary computations throughout application
- Input bytes can flow all over address space
  - Condition typically references input bytes (or derived values) as stored in application-specific data structures
  - Working with compiled data structures encoded in flat address space of stripped binary
- Application works with raw bytes from input file
- But symbolic expression tree uses symbolic input fields

Symbolic Expression Tracing

- Goal – symbolic expression tree for condition
  - Internal nodes are operations (add, sub, cmp, …)
  - Leaves are constants and input bytes
- Execute instrumented application (Valgrind)
- Record expression derivation information
  - Instrument system call I/O operations (record mapping between input bytes and memory)
  - Trace VEX IR operations (add, sub, cmp, mv, …)(record how result derived from operands)

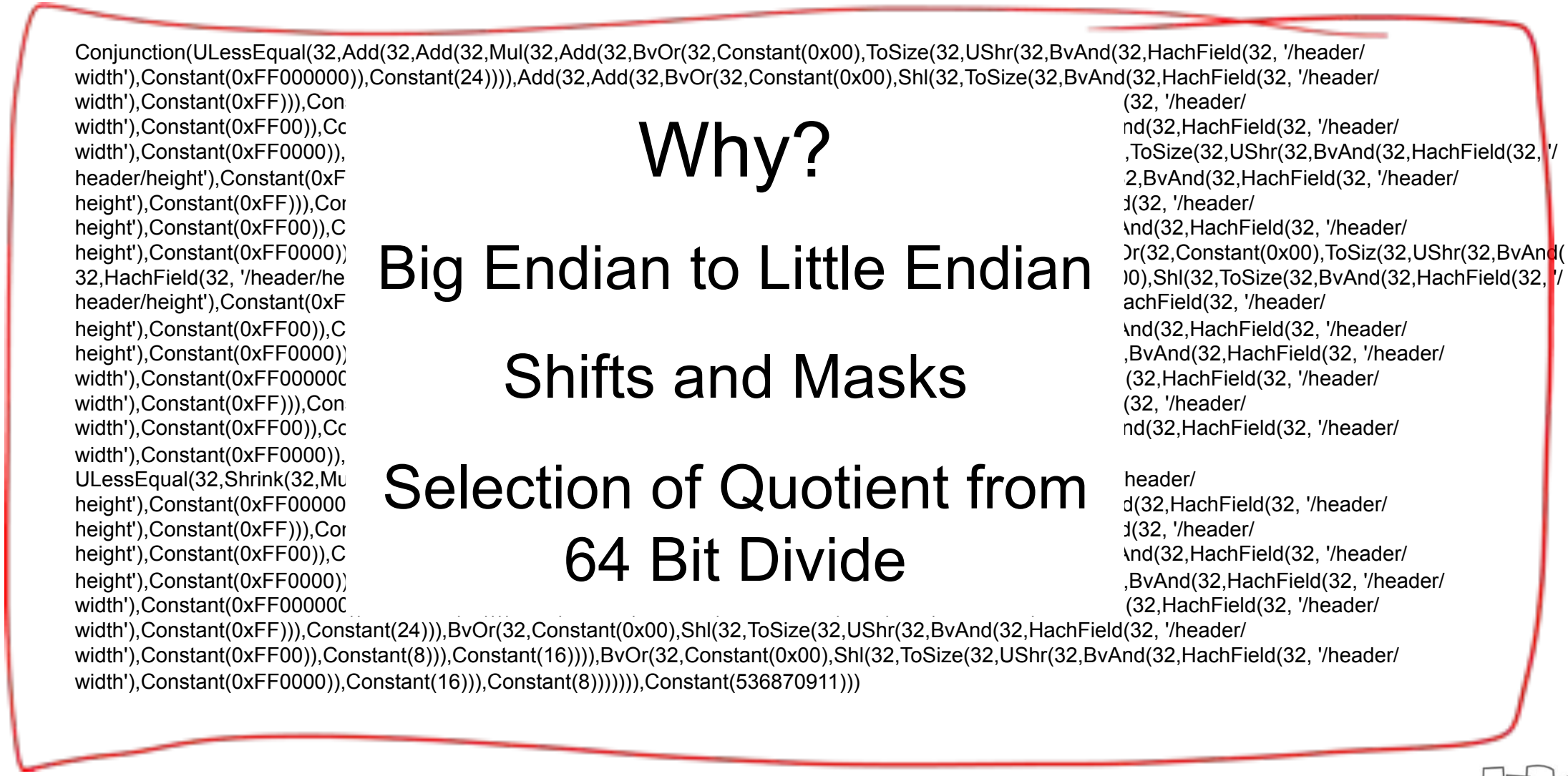
VEX IR From Donor



Is This Plausible?

- Maybe, but we don't do it - too much overhead
- So we use an optimization – two executions
  - First execution: value tracing
    - Record bytes that affect each computed value
    - Determine *relevant bytes* (input bytes that influence condition)
  - Second execution: symbolic expression tracing
  - But only for values that involve relevant bytes

Extracted Condition from viewnior



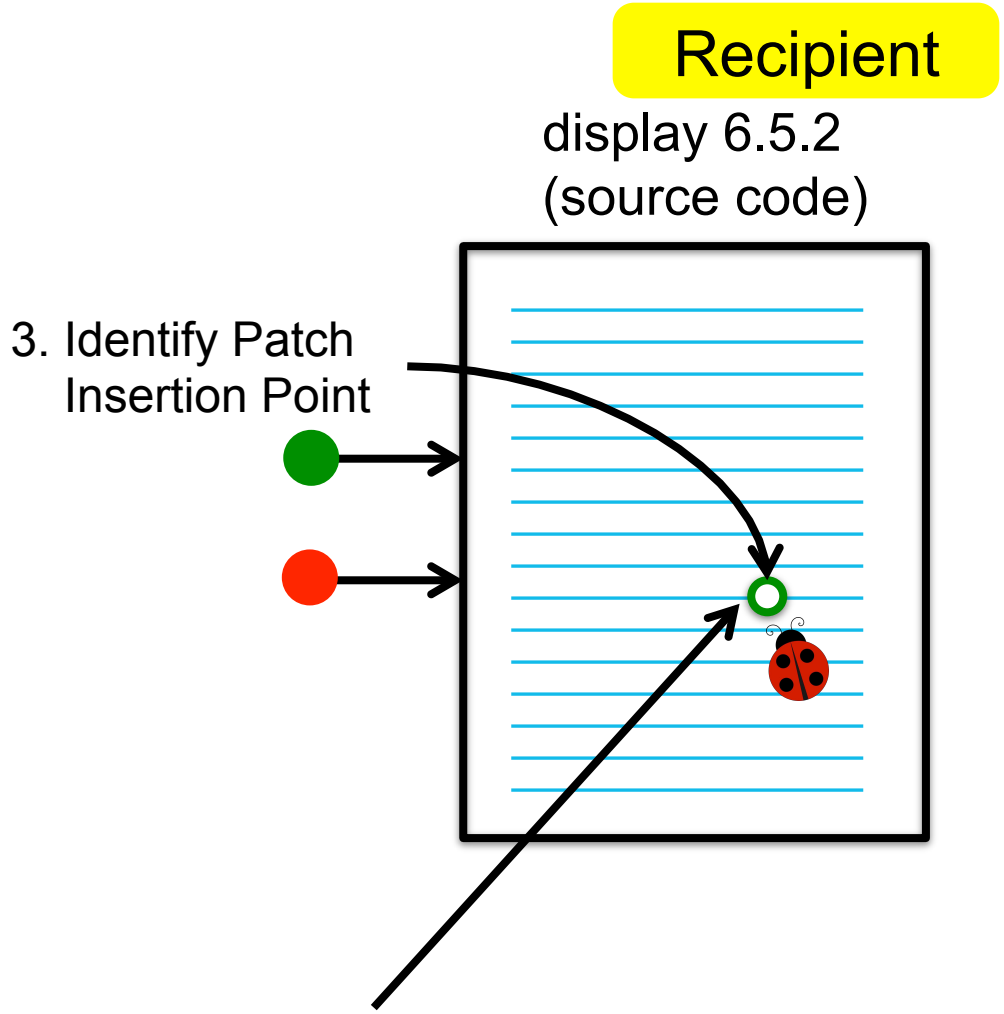
Why?

Big Endian to Little Endian

Shifts and Masks

Selection of Quotient from 64 Bit Divide

Identify Patch Insertion Point



Identify Patch Insertion Point in Recipient

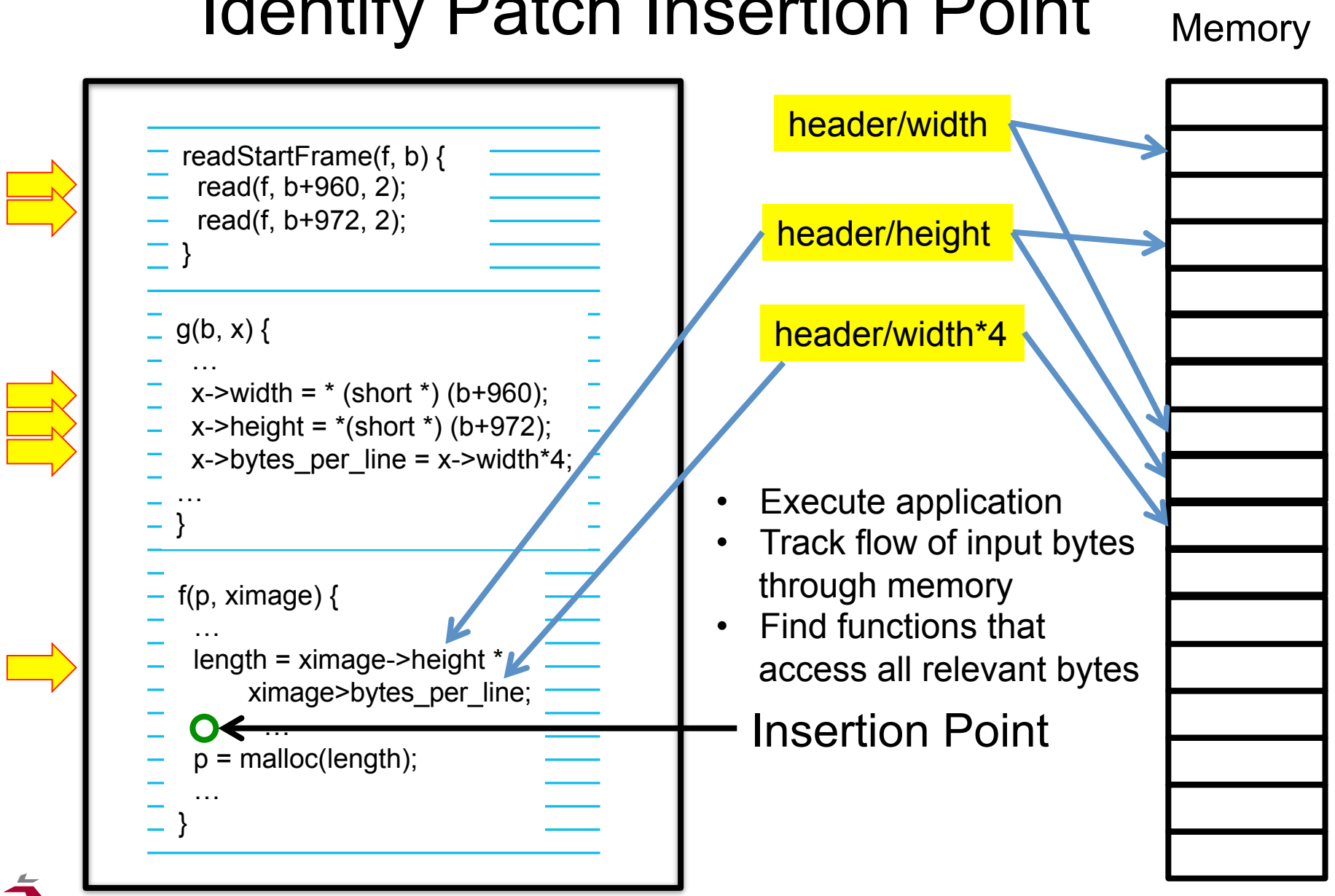
- Key issue: relevant input field values need to be available at patch insertion point
- Execute instrumented recipient
- Trace flow of values through application
  - Trace flow of input fields through memory/values
  - Find functions that access **all** relevant bytes (directly or indirectly via computed values)
  - Program points after last relevant load/store are potential patch insertion points

Identify Patch Insertion Point

Rationale

- If a function accesses relevant input bytes
- Then should be able to find source-level expressions in function for relevant bytes
- And so should be able to generate a source-level patch that uses relevant bytes

Identify Patch Insertion Point

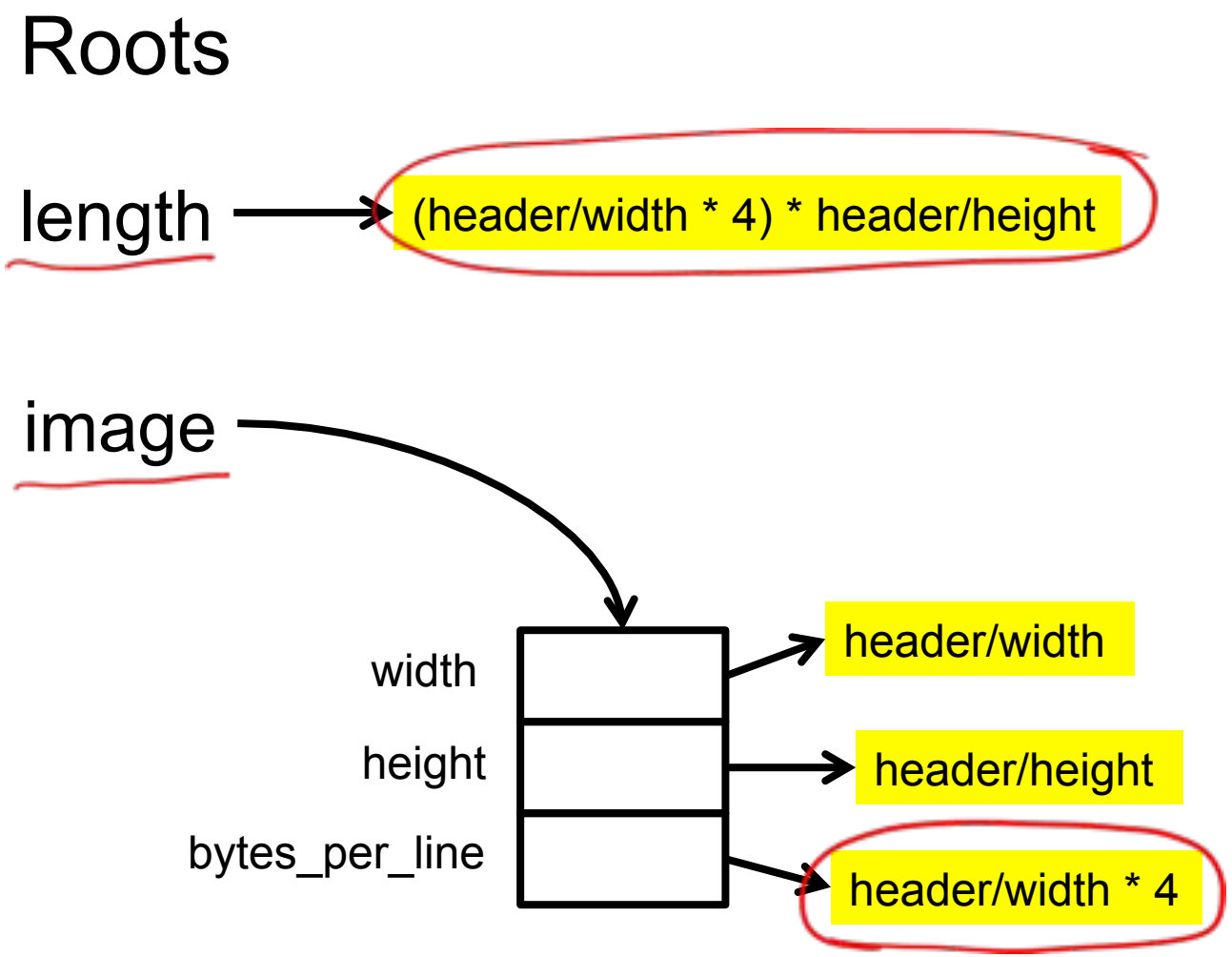


- Execute application
- Track flow of input bytes through memory
- Find functions that access all relevant bytes

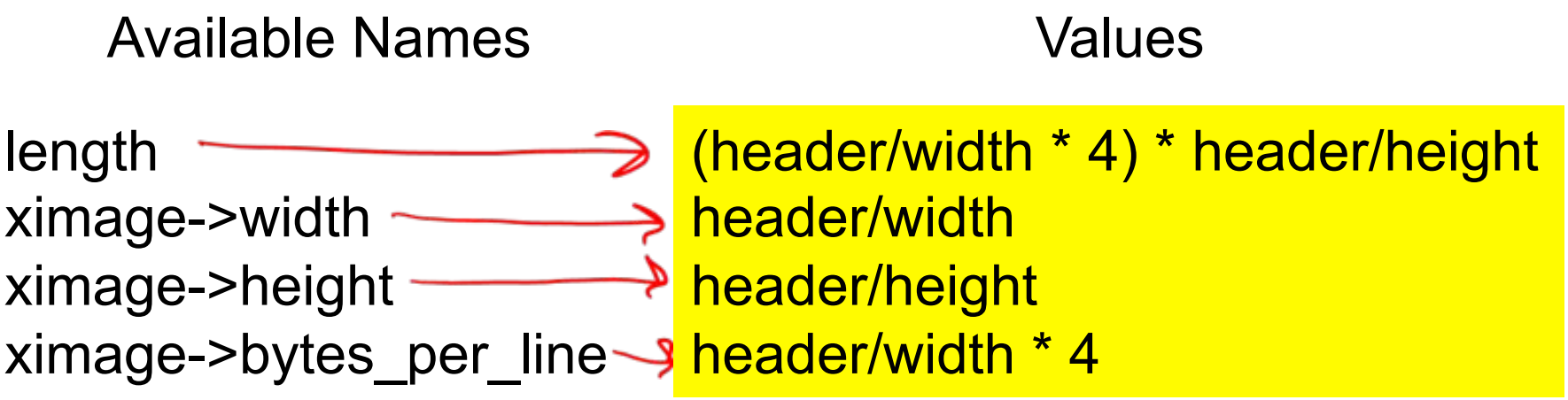
Patch Translation

- Find source-level names for input bytes
- Step 1: Use debugging information to find roots
  - Local variables
  - Global variables
  - Parameters
- Step 2: Traverse from roots, find *available names*
  - Find values that involve relevant bytes
  - Record source-level expressions for those bytes
- Step 3: Use available names to translate patch into source code name space of recipient

Finding Available Names



Available Names



Translate *E* Into Source Code Name Space At Insertion Point

Application-independent representation of check ***E***

Set of source-code variables with input bytes ***Vars***

Directly translate constants

Use SMT solver to find ***V*** with same value as ***E***

Otherwise decompose ***E*** and recurse

```
Parameters:
  E: A symbolic expression
  Vars: A set of active variables
  For each V in Vars, V.var is the variable
name; V.exp is the symbolic expression that
corresponds to the value of the variable.
Return:
  Rewriten expression of E or
  false if failed

Rewrite(E, Vars) {
    if (E is constant)
        return E
    end if
    for V in Vars
        if (SolverEquiv(E,V.exp))
            Ret.opcode ← VAR
            Ret.op1 ← V.var
            return Ret;
        end if
    end for
    if (E.opcode is unary operation)
        Ret.opcode ← E.opcode
        Ret.op1 ← Rewrite(E.op1,Vars)
        if (Ret.op1 != false)
            return Ret
        end if
    else if (E.opcode is binary operation)
        Ret.opcode ← E.opcode
        Ret.op1 ← Rewrite(E.op1,Vars)
        Ret.op2 ← Rewrite(E.op2,Vars)
        if (Ret.op1 ≠ false and
            Ret.op2 ≠ false)
            return Ret
        end if
    end if
    return false
}
```

Final Patch

```
if (!((((uint32_t) ((((uint64_t) (((uint32_t) ((0 | ((uint64_t) length)) |
     (((uint64_t) (length >> ((uint32_t) 31))) << 32)))
      % ximage->bytes_per_line)) << 32) |
     ((uint64_t) (((uint32_t) ((0 | ((uint64_t) length)) |
     (((uint64_t) (length >> ((uint32_t) 31))) << 32)))
      / ximage->bytes_per_line)))) == ximage->height))) {
    printf("Horizontal Code Transfer reject input\n");
    ThrowXWindowFatalException(XServerError,
      "Horizontal Code Transfer Protection", "Test String");
  exit(-1);
}
```

Patch Journey

viewnior source

```
rowstride = width * 4;
bytes = height * rowstride;
if (bytes / rowstride != height) {
    /* overflow */
```

gcc ⇩

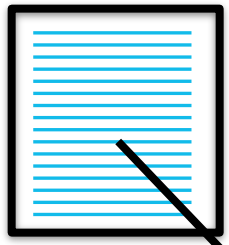
viewnior 1.4

display source

```
if (…) {
    /* overflow */
```

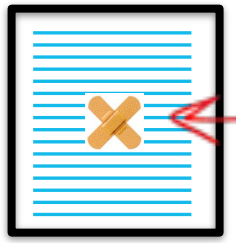
gcc ⇩

Display 6.5.2



Find & Extract

Translate & Insert



Application-Independent Representation of Check

Patch Validation

- Run On Test Inputs
- For Integer Overflow Errors
  - Trace path to error
    - Record necessary conditions
    - Input fields in variables
  - Use SMT solver to find an input that
    - Satisfies necessary conditions
    - Causes integer overflow
  - If no such input exists, patch validates

Tractable Numbers of Candidate Checks and Insertion Points

Application	Format	Error Location	Donor	Patch Generation Time	Candidate Checks	Insertion Points
cwebp 0.3.1	jpeg	jpegdec.c:248	feh 2.9.3	2 min	8	38
			mtpaint 3.4.0	5 min	7	214
			viewnior 1.4	12 min	1	38
dillo 2.1	png	png.c:203	feh 2.9.3	8 min	5	21
			mtpaint 3.4.0	10 min	1	21
			viewnior 1.4	17 min	1	21
		fltkimagebuf.cc:39	feh 2.9.3	4 min	5	33
			mtpaint 3.4.0	10 min	1	33
			viewnior 1.4	13 min	1	33
display 6.5.2	tiff	xwindow.c:5619	feh 2.9.3	8 min	20	74
			viewnior 1.4	15 min	10	74
		cache.c:803	feh 2.9.3	4 min	20	49
			viewnior 1.4	15 min	10	49
swfplay 0.5.5	swf	jpeg_rgb_decoder.c:253	gnash	12 min	8	43
		jpeg.c:192	gnash	25 min	3	38

Tractable Numbers of Candidate Checks and Insertion Points

Application	Format	Error Location	Donor	Patch Gen Time	Candidate Checks	Insertion Points
cwebp 0.3.1	jpeg	jpegdec.c:248	feh 2.9.3	2 min	8	38
			mtpaint 3.4.0	5 min	7	214
			viewnior 1.4	12 min	1	38
dillo 2.1	png	png.c:203	feh 2.9.3	8 min	5	21
			mtpaint 3.4.0	10 min	1	21
			viewnior 1.4	17 min	1	21
		fltkimagebuf.cc:39	feh 2.9.3	4 min	5	33
			mtpaint 3.4.0	10 min	1	33
			viewnior 1.4	13 min	1	33
display 6.5.2	tiff	xwindow.c:5619	feh 2.9.3	8 min	20	74
			viewnior 1.4	15 min	10	74
		cache.c:803	feh 2.9.3	4 min	20	49
			viewnior 1.4	15 min	10	49
swfplay 0.5.5	swf	jpeg_rgb_decoder.c:253	gnash	12 min	8	43
		jpeg.c:192	gnash	25 min	3	38

jasper Buffer Overflow Error

- jasper converts jpeg2K files
- jpeg2K images can have multiple tiles
- jasper has off by one error in tile handling code

Donor

openjpg 1.5.2 source

```
if ((tileno < 0) ||
    (tileno >= (cp->tw*cp->th))) {
  …
  return;
}
```

Recipient

jasper 1.701.0 patch

```
if (!(!(dec->numtiles <= sot->tileno))) {
  exit(-1);
}
```

Correct Check

gif2tiff Buffer Overflow Error

- gif specification defines max code size is 12
- git2tiff does not check
- if code size > 12, buffer overflow

Donor

display 6.5.2-9 source

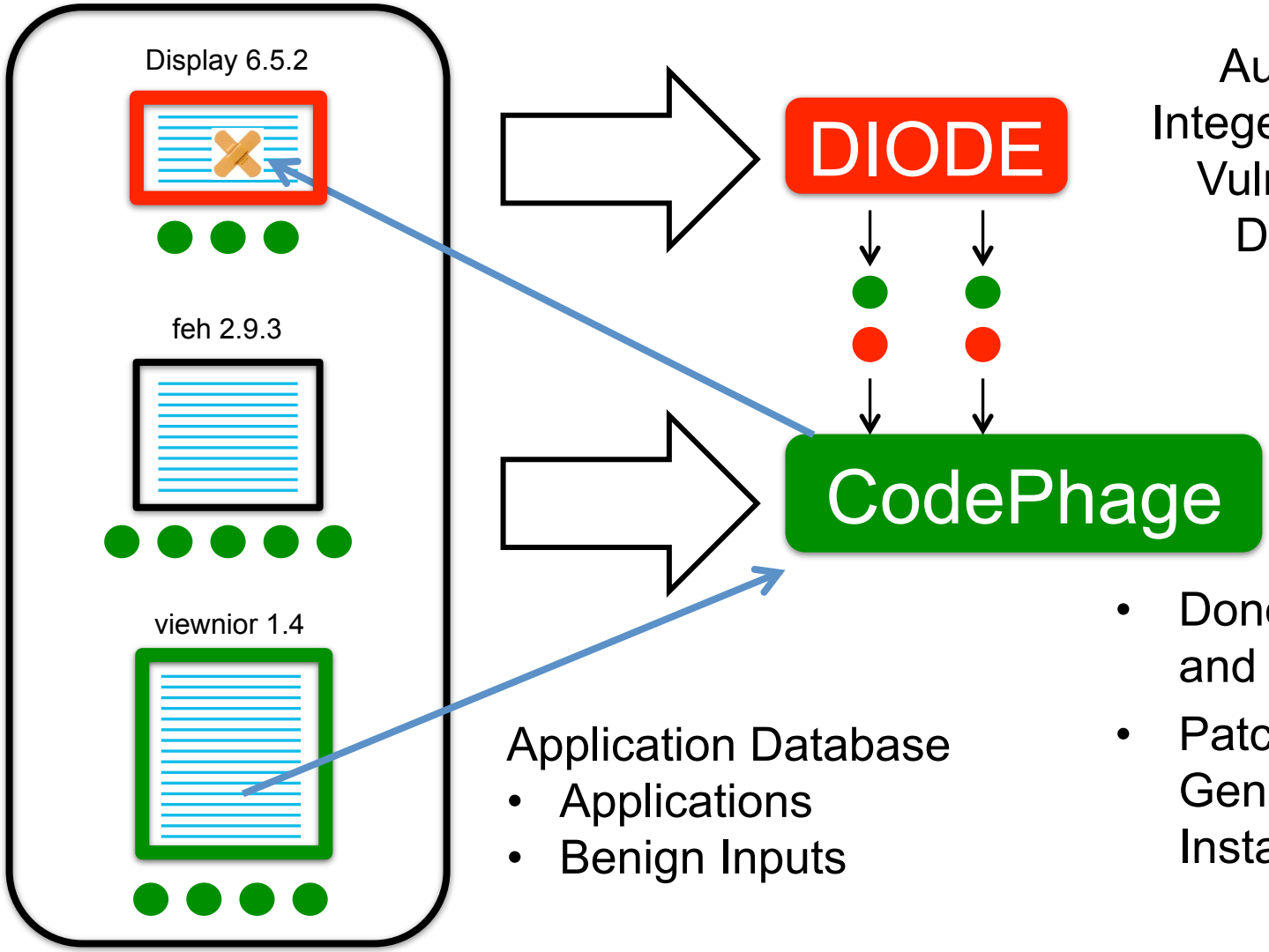
```
#define MaximumLZWBits 12
if (data_size > MaximumLZWBits)
      ThrowBinaryException(…);
```

Recipient

gif2tiff (libtiff 4.0.3) patch

```
if (!(((unsigned char)
       (((unsigned int) datasize) <=
        ((unsigned int) 12))))) {
  exit(-1);
}
```

Bigger Picture



Automatic
Integer Overflow
Vulnerability
Detector

Application Database
- Applications
- Benign Inputs

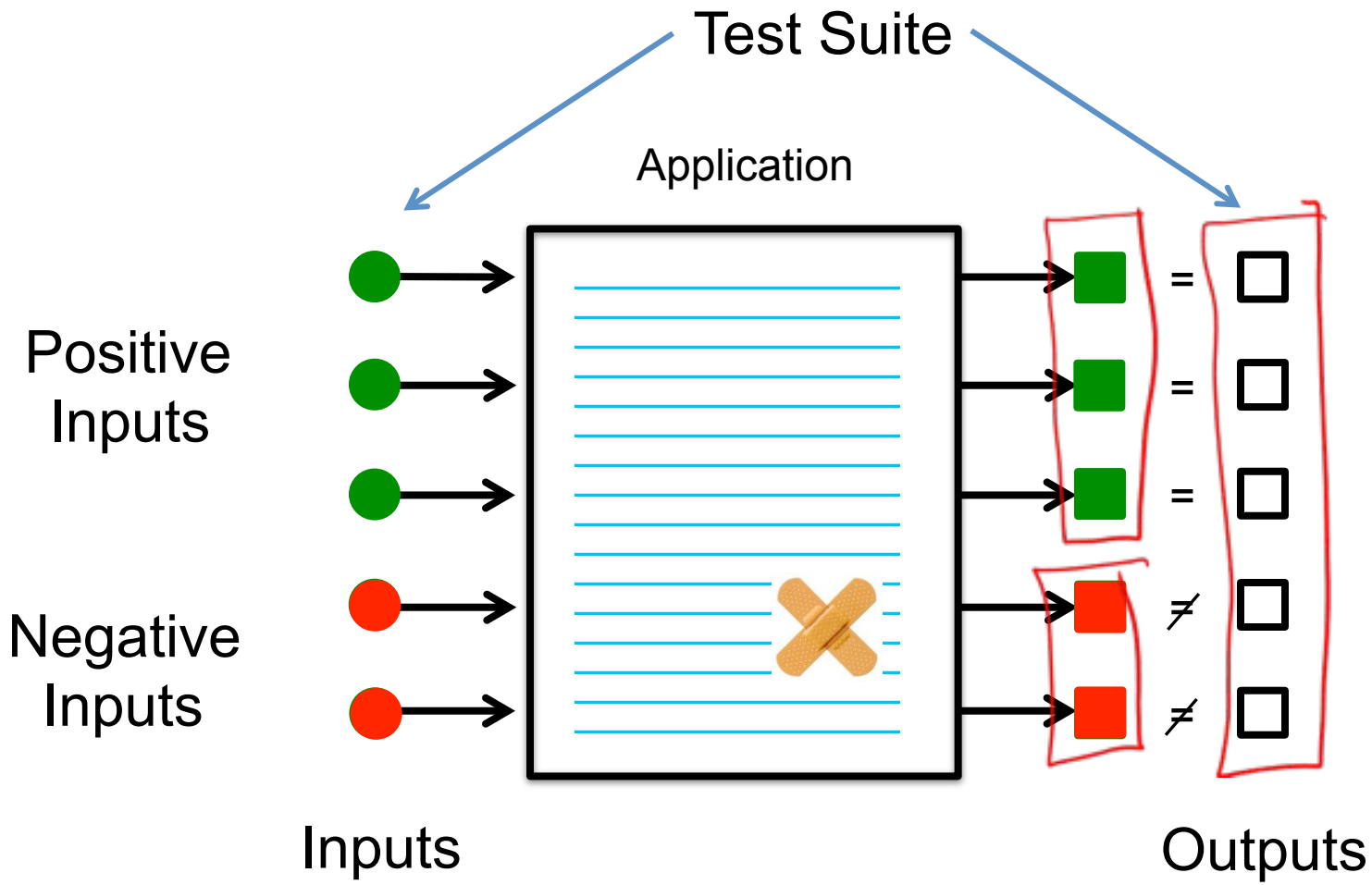
- Donor Evaluation and Selection
- Patch Extraction, Generation, and Installation

RC2: Another Error Patching System

Usage Scenarios

- Improving existing applications
  - Patch software defects
  - Remove security vulnerabilities
- New development approaches
  - Automatic Correctness Acquisition
    - Develop common case code (only)
    - Acquire error detection and handling code from other applications
  - Hybrid Applications
    (combine parts from multiple applications)

RC2 Goal



First Step: Defect Localization

Statement Priority

- Statements executed by negative inputs
- Statements not executed by positive inputs
- Statements executed late in execution

$$N(s) = |\{ i \in \text{Negative. } i \text{ executes } s \}|$$

$$P(s) = |\{ i \in \text{Positive. } i \text{ executes } s \}|$$

$$L(s) = \sum_{i \in Neg} \text{index of last execution of } s$$

Defect Localization Priority

s1 > s2 if

N(s1) > N(s2) or

N(s1)=N(s2) and P(s1) < P(s2) or

N(s1)=N(s2) and P(s1)=P(s1) and L(s1) > L(s2)

N(s) = | { i∈ Negative. i executes s } |

P(s) = | { i∈ Positive. i executes s } |

$L(s) = \sum_{i \in Neg}$ index of last execution of s

Using Defect Localization Priority

- Priority of
  if (C) { S1 } else { S2 }
  is maximum priority of **S1, S2**
- Search space is 500 highest priority statements

Condition Synthesis

- Defect localizer identifies an if statement

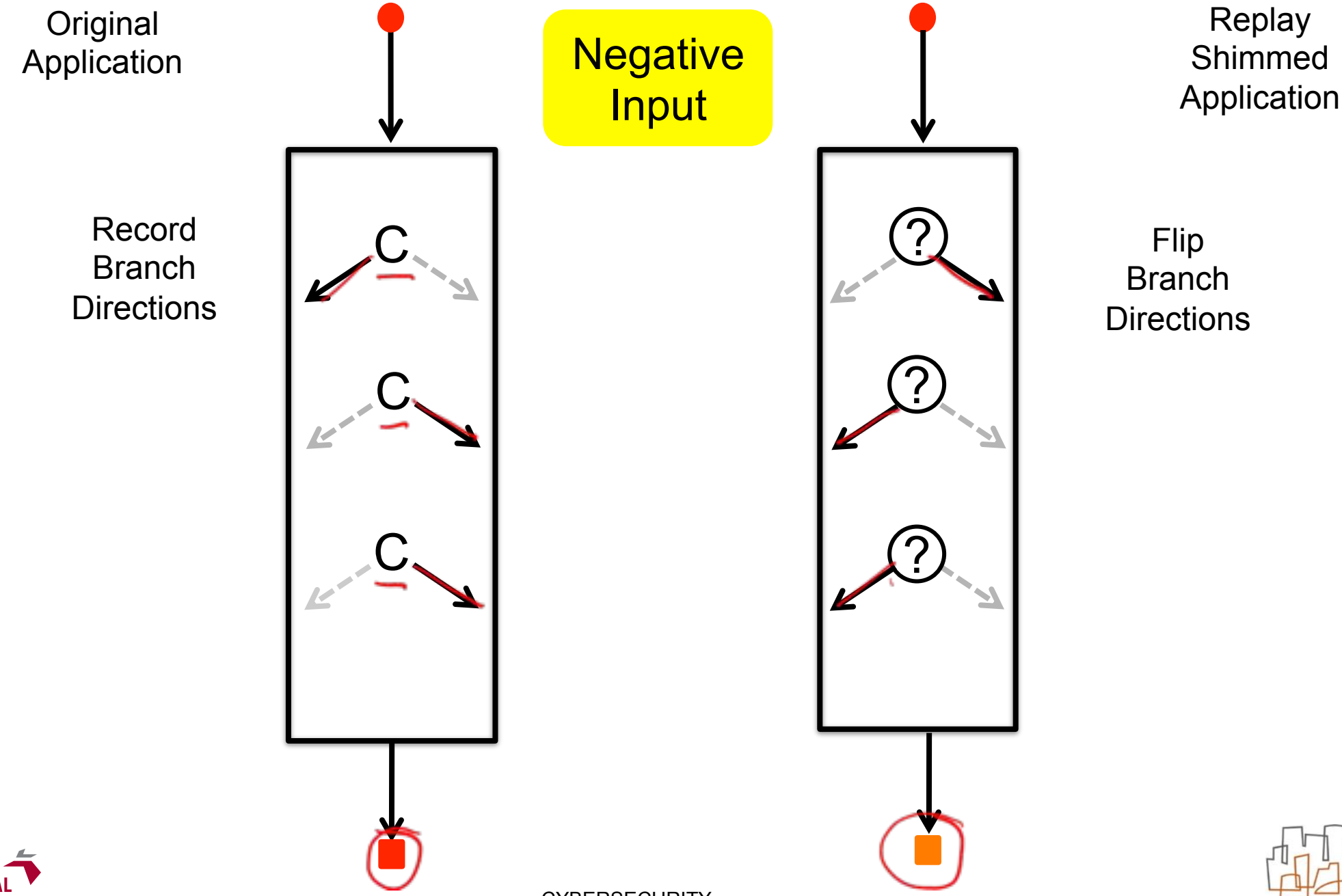
  if (C) { … } else { … }

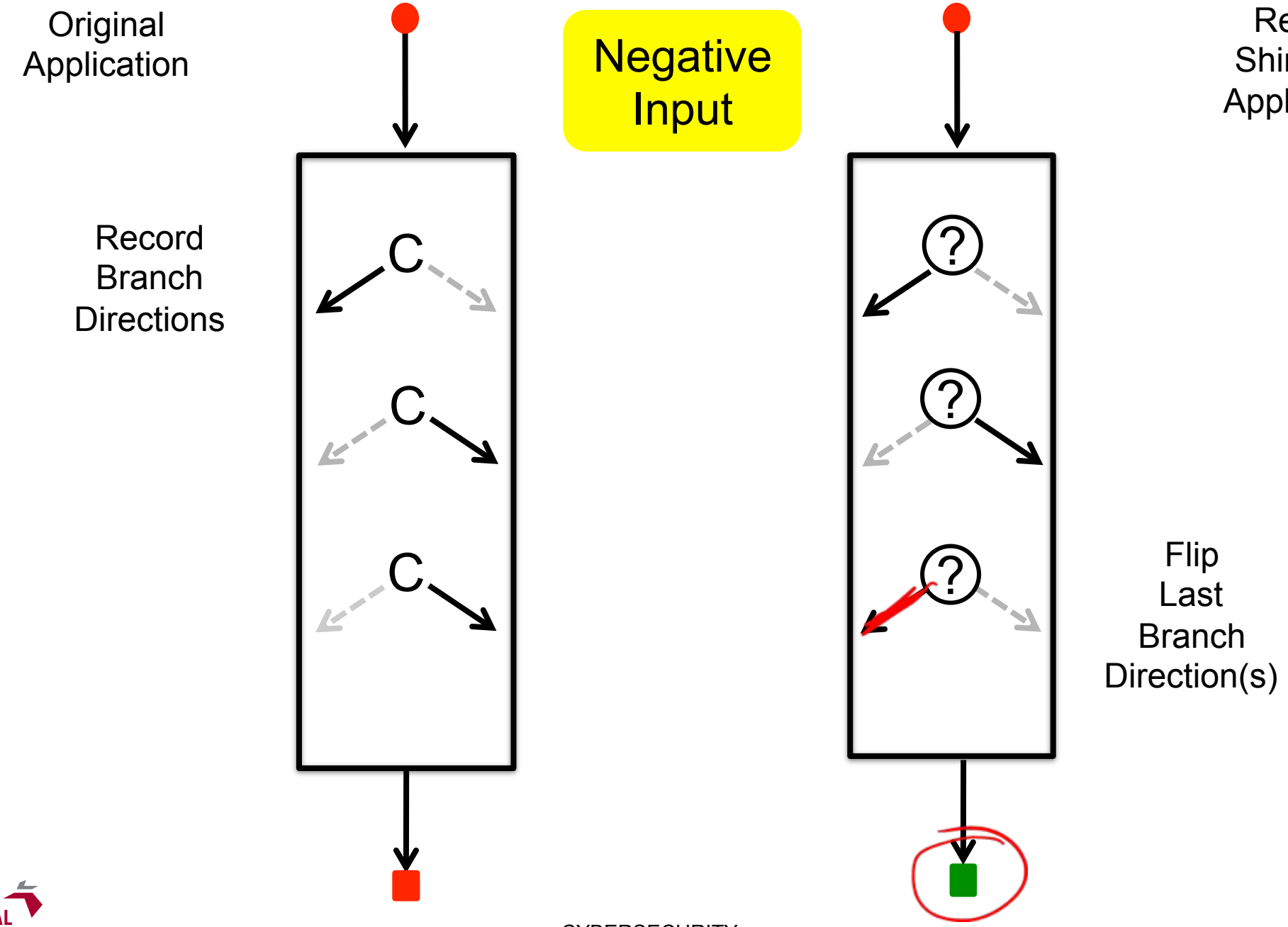
- Consider two kinds of patches

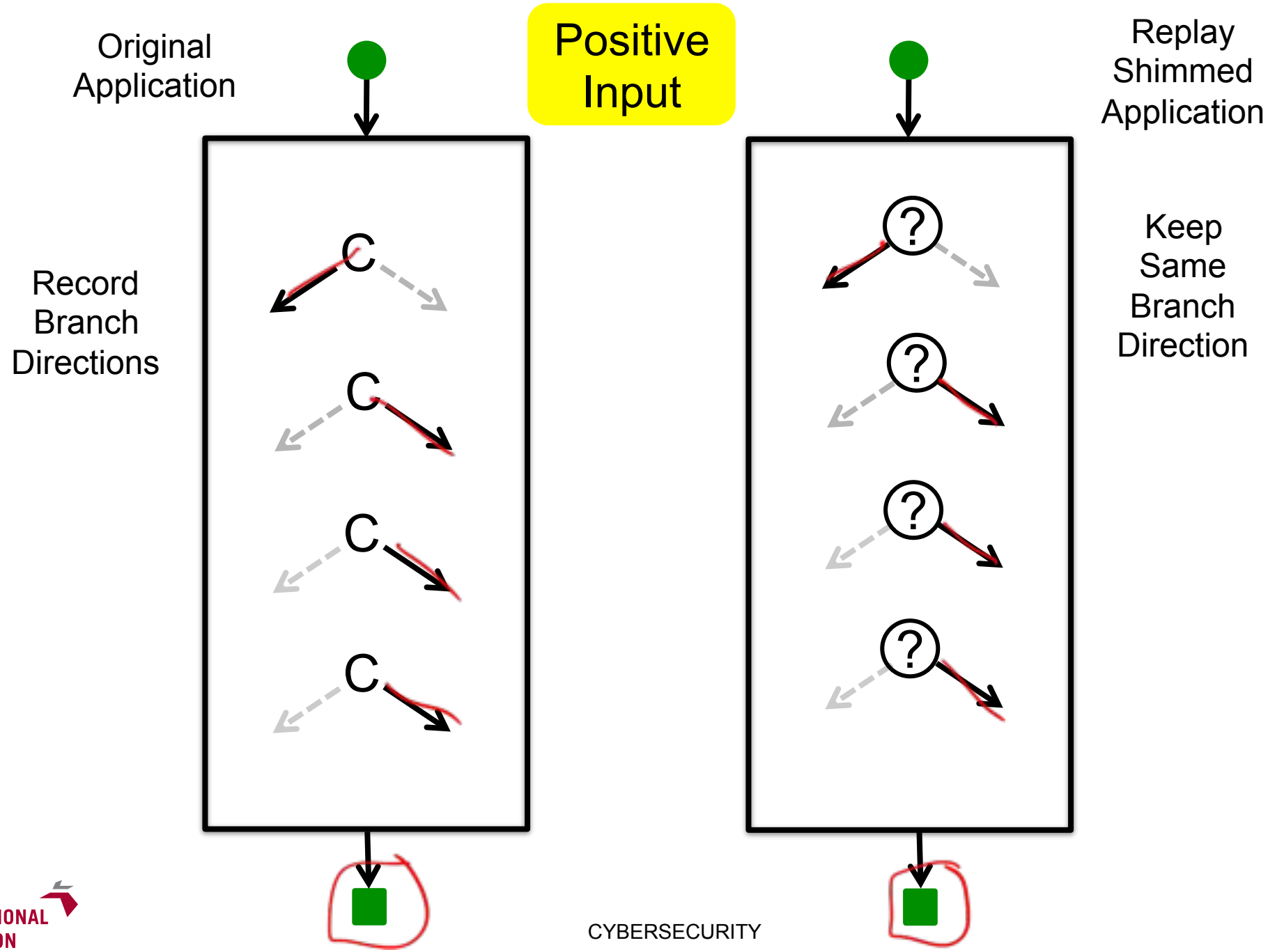
  if (C || **E**) { … } else { … } (loosen)

  if (C && **E**) { … } else { … } (tighten)

- Two steps
  - Identify branch direction sequence that produces correct output on all inputs
  - Synthesize **E** that generates (close to) that sequence







Condition Synthesis (C is x > 4)

E is y = 0

Relevant Variables

x	y	z	x > 4 Original Branch Direction	x > 4 || E Desired Branch Direction
5	1	3	T	T
2	1	7	F	F
7	1	5	T	T
9	0	2	T	T
6	2	3	T	T
1	3	9	F	F
0	0	6	**F**	**T**

Branch Directions From Positive Input

Branch Directions From Negative Input

Goal: Synthesize an **E** that provides desired branch directions

Building On Condition Synthesis

- Defect localizer identifies **S**
- Guard insertion mutation:

  S ⟶ **if (E) { S }**

  Use condition synthesis to generate **E**

- Conditional control-flow insertion mutation:

  S ⟶ **if (E) return C; S**

  S ⟶ **if (E) goto label; S**

  S ⟶ **if (E) break; S**

  Use condition synthesis to generate **E**

Another Compound Mutation

- Defect localizer identifies **S**
- Replace subexpression mutation:

  Variable replacement: **S** ⟹ **S[v1/v2]**

  Constant replacement: **S** ⟹ **S[c1/c2]**

  Function replacement:

  S ⟹ **S[f(e1, …, en)/g(e1, …, en)]**

Yet Another Compound Mutation

- Defect localizer identifies **S**
- Add statement via copy and replace
  - Choose some statement **Q** in program
  - Copy and Replace **Q** before **S**:

Variable replacement: **S → Q[v1/v2]; S**

Constant replacement: **S → Q[c1/c2]; S**

Function replacement:

S → Q[f(e1, …, en)/g(e1, …, en)]; S

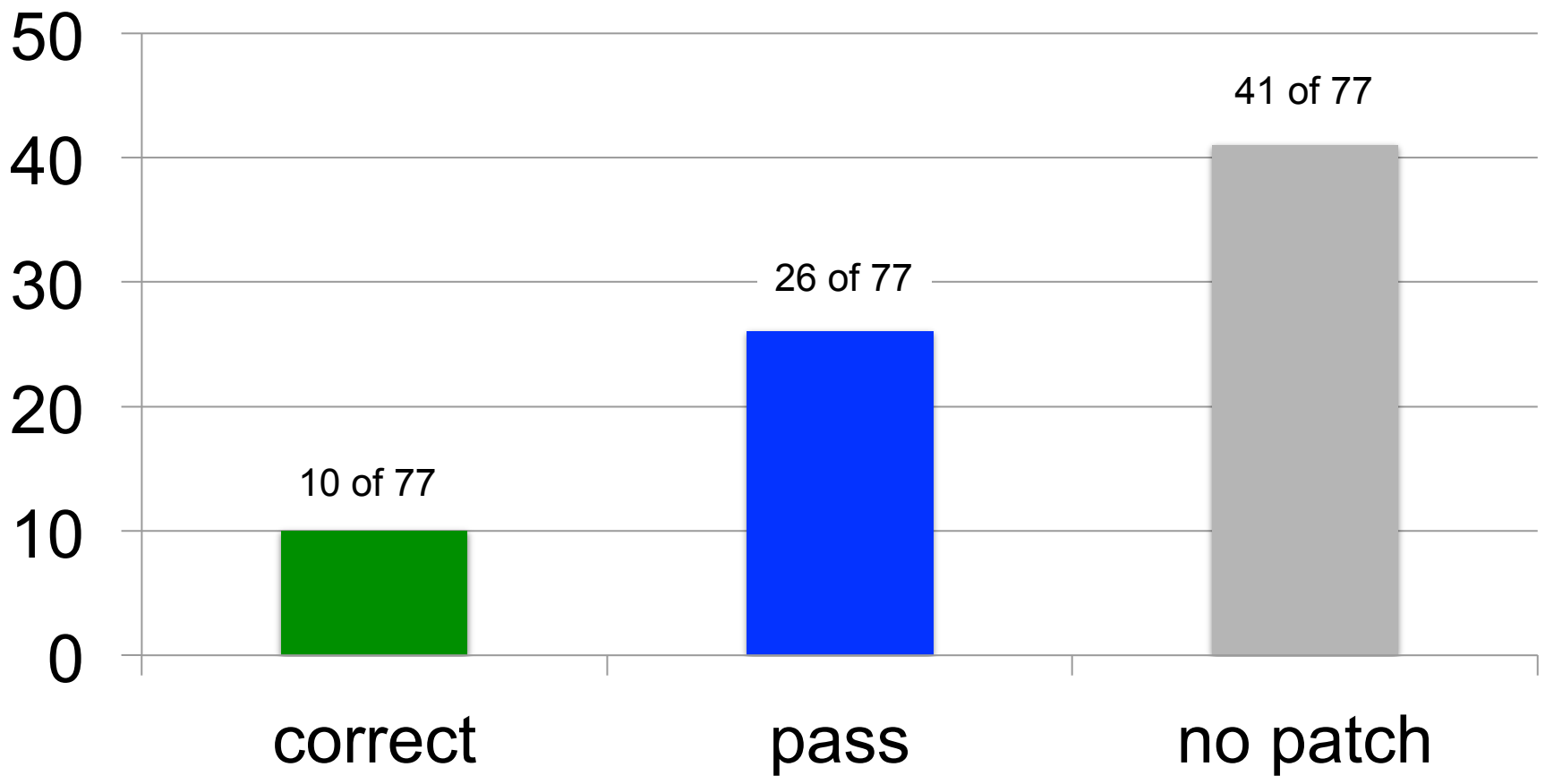
Search Space Possibilities

- One compound mutation only
  - Simple space, relatively efficient to search
  - Targets small patches
- Multiple combined compound mutations
  - Richer space of candidate patches
  - Less efficient to search
- Current system: one compound mutation only

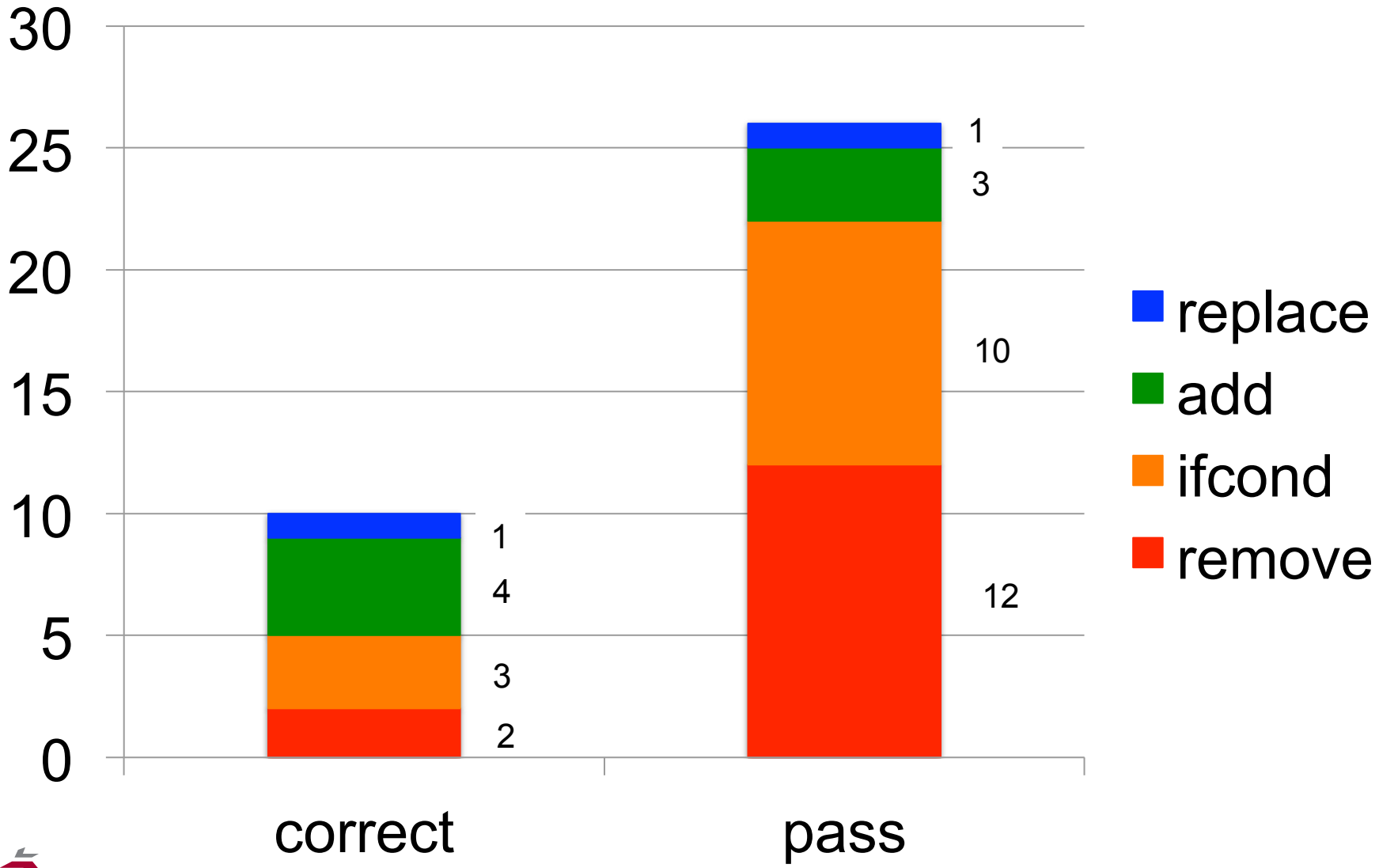
Results

- Implemented RC2
- Evaluated on 77 defects from 3 applications
  - libtiff (24 defects)
  - lighttpd (9 defects)
  - php (44 defects)
- Each defect comes with
  - Test suite that exposes defect
  - Version with developer fix applied (sometimes this is an earlier version)

Defect Outcomes



Patch Breakdown



A Cautionary Example

- Defective integer overflow check:

```
/* Check for overflow. */
if (!dir->tdir_count || !w || (tsize_t)dir->tdir_count / w != cc)
        goto bad;
```

  - One test case checks for incorrect input rejection
  - No test case checks for correct input rejection
- Patch simply removes check
- Reintroduces security vulnerability (CVE-2006-2025)
- Pattern: removal of unprotected functionality

Why Not More Correct Patches? Two Possibilities:

1. Search space does not contain correct patch
2. Weak test suite
   - Search space does contain correct patch
   - But an earlier incorrect patch passes test suite

How Many More Correct Patches Exist In Search Space?

5

Defect	Search Space	Correct Patch At	Max Test Cases	Wrong Location	Wrong Synthesis
php-A	10347	355	3	2	1
php-B	3431	22	5	0	5
php-C	5028	1996	-	-	-
libtiff-A	64821	131	13	0	13
libtiff-B	71359	7354	-	-	-

Reasons for RC2 Success

- Separate condition synthesis from main search algorithm
  - Significantly reduces patch search space
  - Focuses search on productive part of space
- Compound mutations with replacement
  - Rich, structured search space
  - Focuses search on productive part of space

Two Kinds of Systems

- General
  - Basic mutations (copy, replace, remove)
  - Generate search space from multiple mutations
  - Try to find mutation that passes test suite
- Targeted
  - Identify class of errors, build technique to eliminate error
  - FOC (out of bounds accesses) [OSDI 2004]
  - ClearView (invariant violations) [SOSP 2009]
  - Bolt (infinite loops) [OOPSLA 2012]
  - RCV (null dereference, divide by zero) [PLDI 2014]
  - PAR (human identified error patterns) [ICSE 2013]

Pros and Cons

General

- Pro
  - Expressive search space
  - Many potential patches
- Con
  - Search space is very large
  - Sparse patch occurrence
  - Difficult to find patches
  - Functionality removal if not protected by test case

Targeted

- Pro
  - No (or limited) search
  - More predictable consequences
  - Hot recovery
  - No source code
- Con
  - Less general

RC2

- More structured than general systems
  - Condition synthesis
  - Compound mutations
  - More effective search space
- Desired functionality must be protected by positive test cases
- Targeted (right now) to small patches

Looking To The Future

- We are now starting to automate traditional software engineering tasks
  - Finding and eliminating defects
  - Transferring code between applications
- Software development today
  - Manual and slow
  - Expensive
- Starting to see how automation may qualitatively transform software our society can produce
  - Reliability and security
  - Functionality

THANK YOU

Martin Rinard

Professor MIT EECS, MIT CSAIL