Constraint Solving in Symbolic Execution

Cristian Cadar

Department of Computing

Imperial College London

SOFTWARE RELIABILITY GROUP

Imperial College London

Invited talk at SMT 2015

18 July, San Francisco, CA, USA

Dynamic Symbolic Execution

- Dynamic symbolic execution is a technique for ***automatically exploring paths*** through a program
  - Determines the feasibility of each explored path using a ***constraint solver***
  - Checks if there are ***any*** values that can cause an error on each explored path
  - For each path, can generate a ***concrete input triggering the path***

Dynamic Symbolic Execution

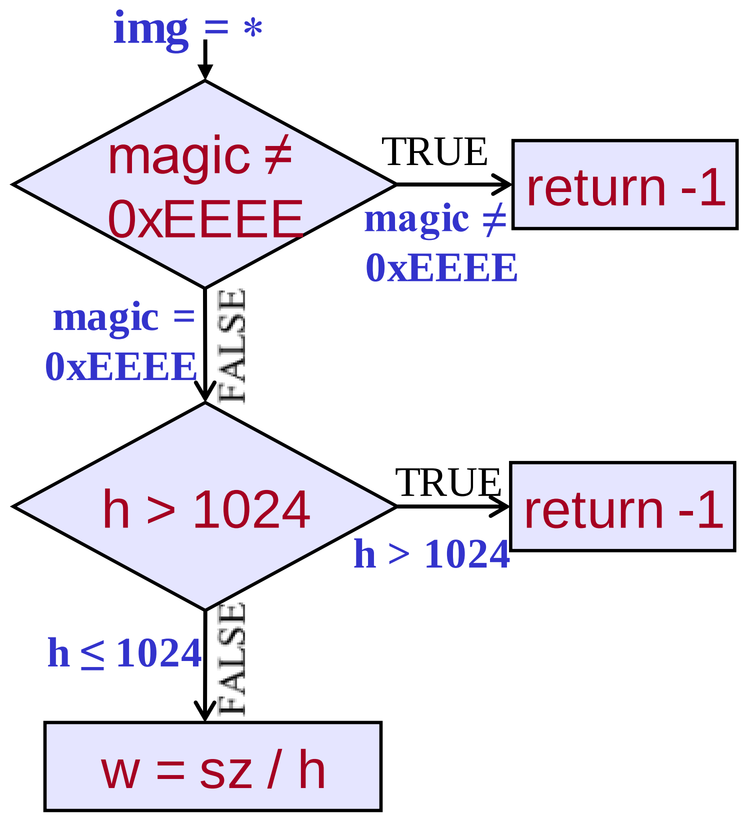
- Received significant interest in the last few years
- Many dynamic symbolic execution/concolic tools available as open-source:
  - **CREST, KLEE, SYMBOLIC JPF**, etc.
- Started to be adopted/tried out in the industry:
  - Microsoft (**SAGE, PEX**)
  - NASA (**SYMBOLIC JPF, KLEE**)
  - Fujitsu (**SYMBOLIC JPF, KLEE/KLOVER**)
  - IBM (**APOLLO**)
  - etc. etc.

Symbolic Execution for Software Testing in Practice: Preliminary Assessment. Cadar, Godefroid, Khurshid, Pasareanu, Sen, Tillmann, Visser, **[ICSE Impact 2011]**

Toy Example

```
struct image_t {
    unsigned short magic;
    unsigned short h, sz;
    ...
```

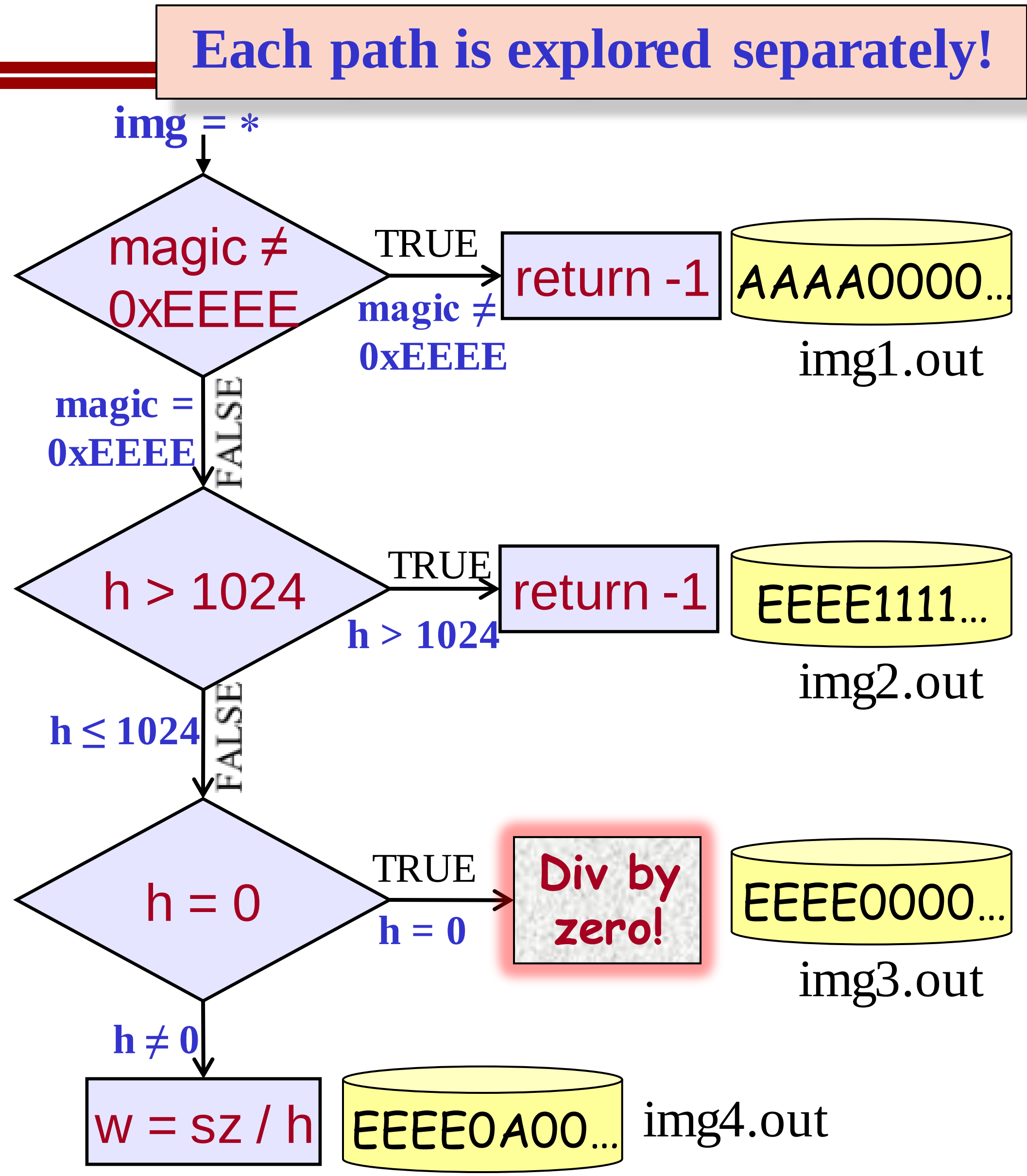
```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



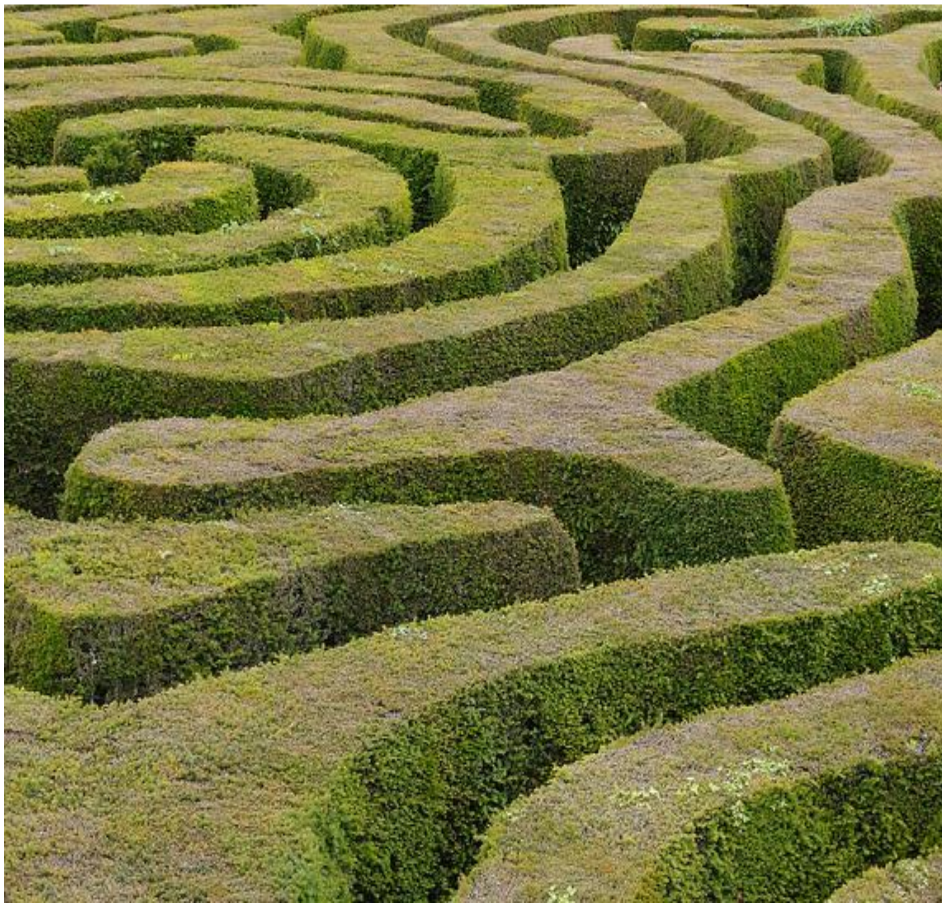
Toy Example

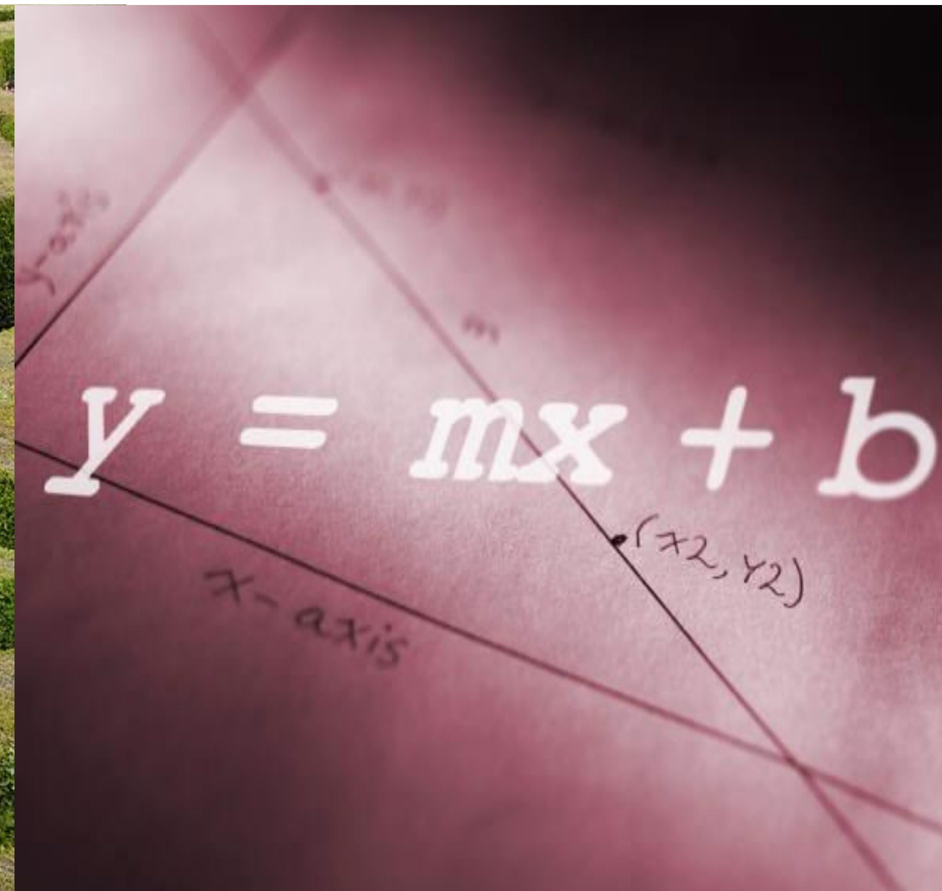
```
struct image_t {
    unsigned short magic;
    unsigned short h, sz;
    ...
```

```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



Scalability Challenges





Rest of the talk

Constraint solving in symex for:

(1) Bug-finding in systems and security-critical code

(2) Recovery of broken documents

(3) Testing and bounded verification of program optimisations (if time)

Bug-Finding

Joint work with:

Daniel Dunbar, Dawson Engler [OSDI 2008]

Junfeng Yang, Can Sar, Paul Twohey, Dawson Engler [IEEE S&P 2008]

Paul Marinescu [ICSE 2012]

Hristina Palikareva [CAV 2013]

JaeSeung Song, Peter Pietzuch [IEEE TSE 2014]

Bug Finding with EGT, EXE, KLEE:
Focus on Systems and Security Critical Code

	Applications
Text, binary, shell and file processing tools	GNU Coreutils, findutils, binutils, diffutils, Busybox, MINIX (~500 apps)
Network servers	Bonjour, Avahi, udhcpd, lighttpd, etc.
Library code	libdwarf, libelf, PCRE, uClibc, etc.
File systems	ext2, ext3, JFS for Linux
Device drivers	pci, lance, sb16 for MINIX
Computer vision code	OpenCV (filter, remap, resize, etc.)
OpenCL code	Parboil, Bullet, OP2

- Most bugs fixed promptly

Coreutils Commands of Death

`md5sum -c t1.txt`	`pr -e t2.txt`
`mkdir -Z a b`	`tac -r t3.txt t3.txt`
`mkfifo -Z a b`	`paste -d\\ abcdefghijklmnopqrstuvwxyz`
`mknod -Z a b p`	`ptx -F\\ abcdefghijklmnopqrstuvwxyz`
`seq -f %0 1`	`ptx x t4.txt`
``printf %d ` ``	`cut -c3-5,8000000- --output-d=: file`

t1.txt:	`\t \tMD5(`	*t3.txt:*	`\n`
t2.txt:	`\b\b\b\b\b\b\b\t`	*t4.txt:*	`A`

[OSDI 2008, ICSE 2012]

Disk of Death (JFS, Linux 2.6.10)

Offset	Hex Values							
00000	0000	0000	0000	0000	0000	0000	0000	0000
. . .				. . .				
08000	464A	3135	0000	0000	0000	0000	0000	0000
08010	1000	0000	0000	0000	0000	0000	0000	0000
08020	0000	0000	0100	0000	0000	0000	0000	0000
08030	E004	000F	0000	0000	0002	0000	0000	0000
08040	0000	0000	0000	. . .				

- **64th sector of a 64K disk image**
- **Mount it and PANIC your kernel**

[IEEE S&P 2008]

Packet of Death (Bonjour)

Offset	Hex Values							
0000	0000	0000	0000	0000	0000	0000	0000	0000
0010	003E	0000	4000	FF11	1BB2	7F00	0001	E000
0020	00FB	0000	14E9	002A	0000	0000	0000	0001
0030	0000	0000	0000	055F	6461	6170	045F	7463
0040	7005	6C6F	6361	6C00	000C	0001		

- **Causes Bonjour to abort, potential DoS attack**
- **Confirmed by Apple, security update released**

[IEEE TSE 2014]

Constraint Solving: Accuracy

- Bit-level modeling of memory is critical in C code
  - Many bugs and security vulnerabilities could only be found if we reason about arithmetic overflows, type conversions, etc.
- Mirror the (lack of) type system in C
  - Model each memory block as an array of 8-bit BVs
  - Bind types to expressions, not bits
- Need a QF_ABV solver
  - We mainly use STP

Constraint Solving: Speed

- Real program generate complex queries
- Queries performed at every branch

To be effective, DSE needs to explore lots of paths → solve lots of queries, <u>fast</u>

Some Constraint Solving Statistics

Application	Instrs/s	Queries/s	Solver %
[	695	7.9	97.8
base64	20,520	42.2	97.0
chmod	5,360	12.6	97.2
comm	222,113	305.0	88.4
csplit	19,132	63.5	98.3
dircolors	1,019,795	4,251.7	98.6
echo	52	4.5	98.8
env	13,246	26.3	97.2
factor	12,119	22.6	99.7
join	1,033,022	3,401.2	98.1
ln	2,986	24.5	97.0
mkdir	3,895	7.2	96.6
Avg:	**196,078**	**675.5**	**97.1**

1h runs using KLEE with STP, in DFS mode

UNIX utilites (and many other benchmarks)

- Large number of queries
- Most queries <0.1s
- Typical timeout: 30s
- Most time spent in the solver (before and after optimizations!)

[CAV'13]

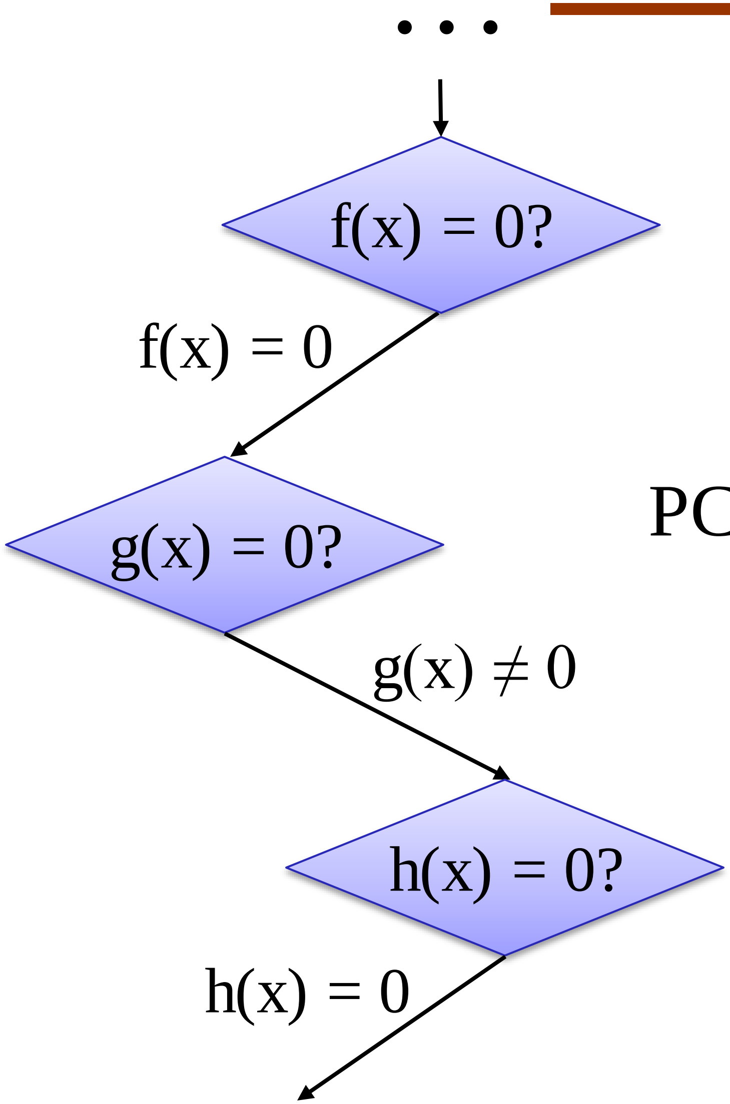
Constraint Solving Performance

We already benefit from the optimisations performed by SAT and SMT solvers

Essential to exploit the characteristics of the constraints generated during symex, e.g.:

1) Conjunctions of constraints
2) Path condition (PC) always satisfiable
3) Large sequences of (similar) queries
4) Must generate counterexamples

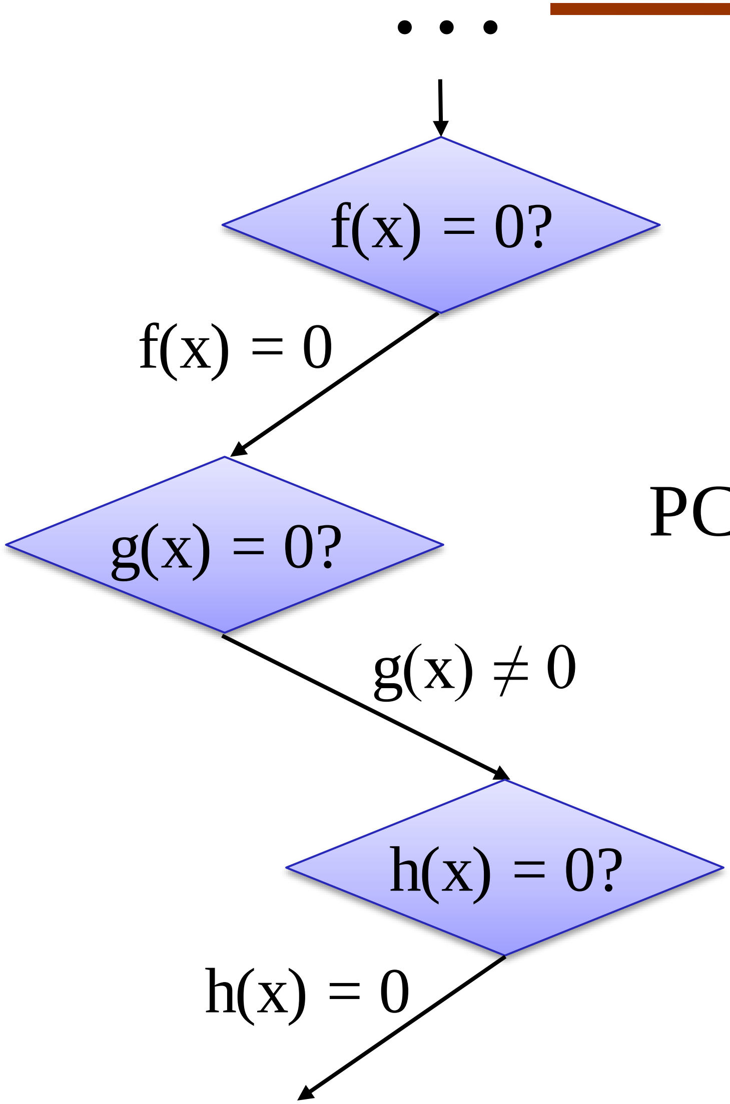
1) Conjunction of constraints



- We explore one path at a time

PC: $f(x) = 0 \ \wedge\ g(x) \neq 0 \ \wedge\ h(x) = 0$

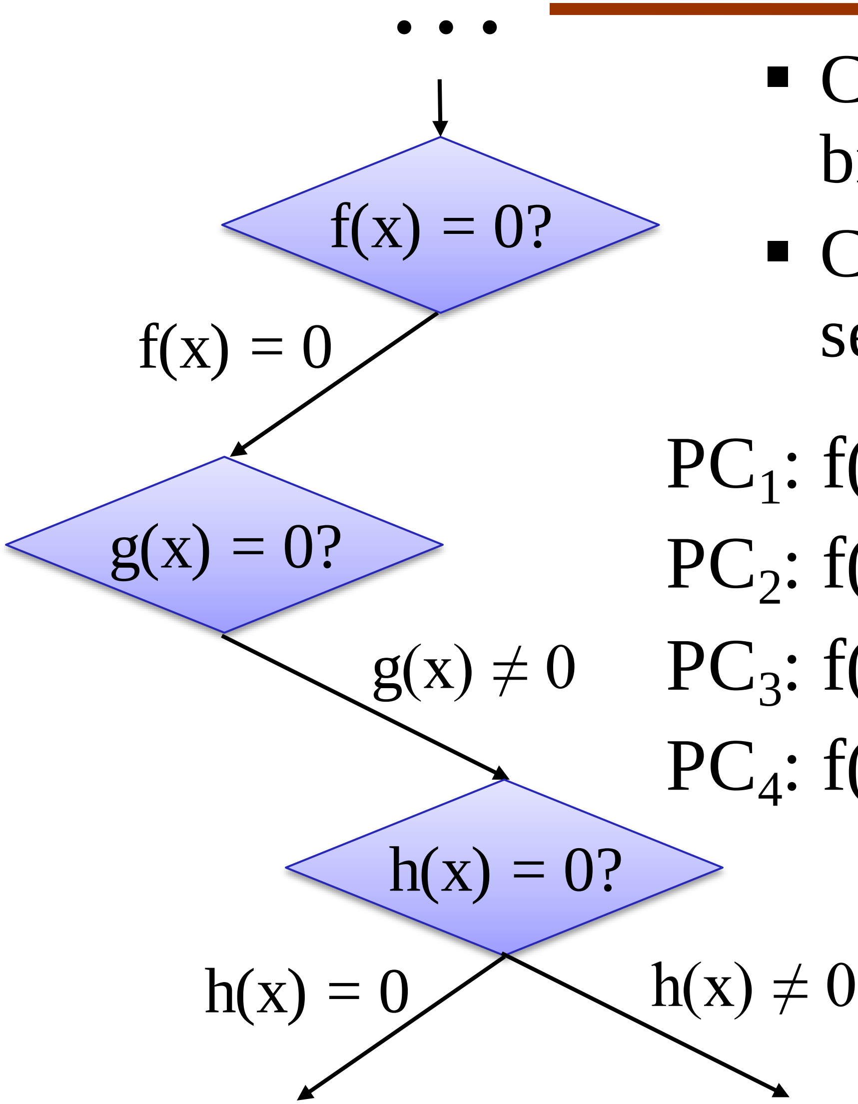
2) PC always satisfiable



- We check for satisfiability at each branch
- We only explore feasible paths

PC: $f(x) = 0 \land g(x) \neq 0 \land h(x) = 0$

3) Large sequence of (similar) queries



- Check for satisfiability at each branch
- Constraints obtained from a fixed set of static branches

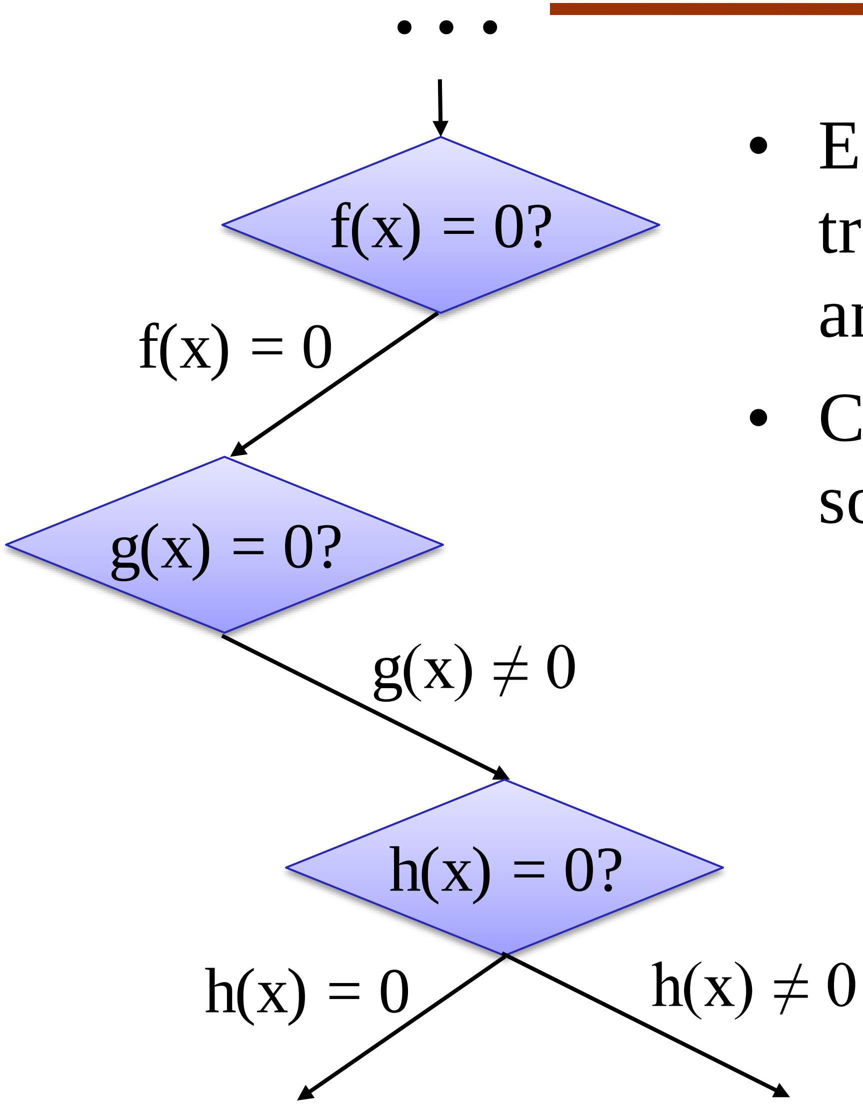
PC_1: $f(x) = 0$

PC_2: $f(x) = 0 \;\wedge\; g(x) \neq 0$

PC_3: $f(x) = 0 \;\wedge\; g(x) \neq 0 \;\wedge\; h(x) = 0$

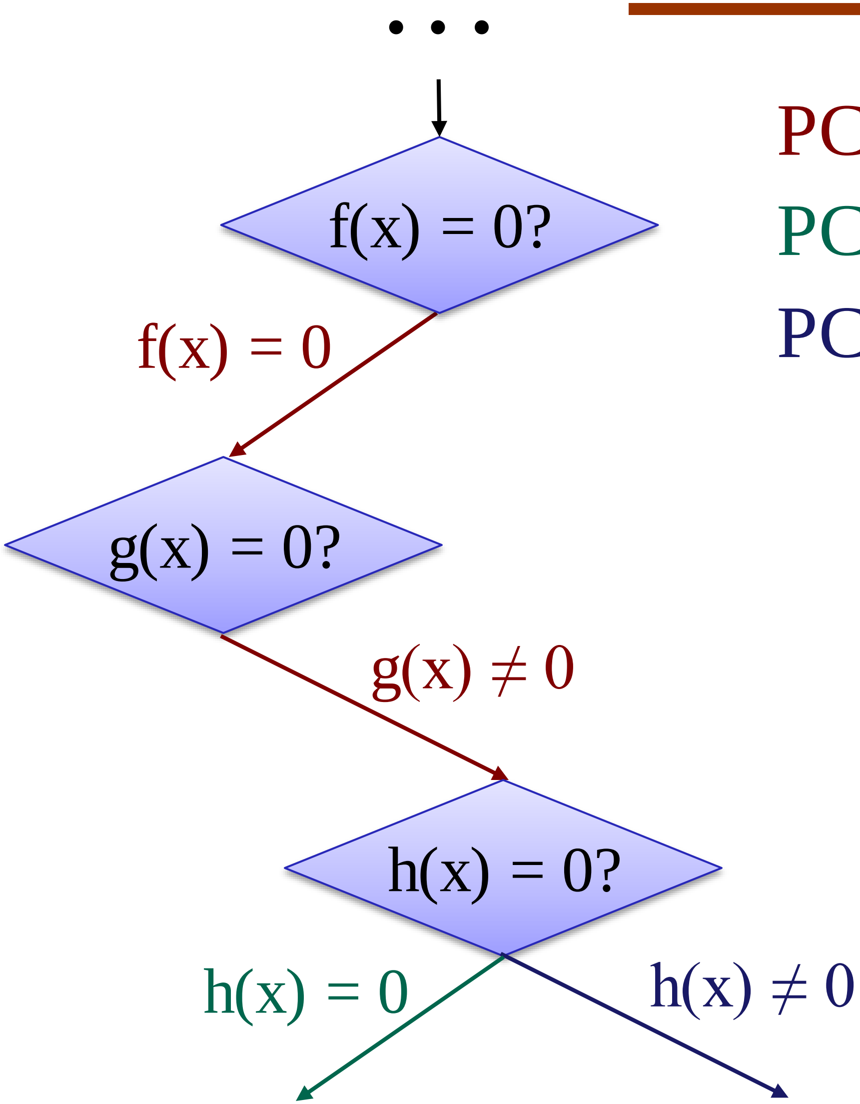
PC_4: $f(x) = 0 \;\wedge\; g(x) \neq 0 \;\wedge\; h(x) \neq 0$

4) Must generate counterexamples



- Essential for reproducing bugs, transitioning between symbolic and concrete
- Can also be exploited for faster solving

Example optimisation



PC_a: $f(x) = 0 \wedge g(x) \neq 0$

PC_b: $f(x) = 0 \wedge g(x) \neq 0 \wedge h(x) = 0$

PC_c: $f(x) = 0 \wedge g(x) \neq 0 \wedge h(x) \neq 0$

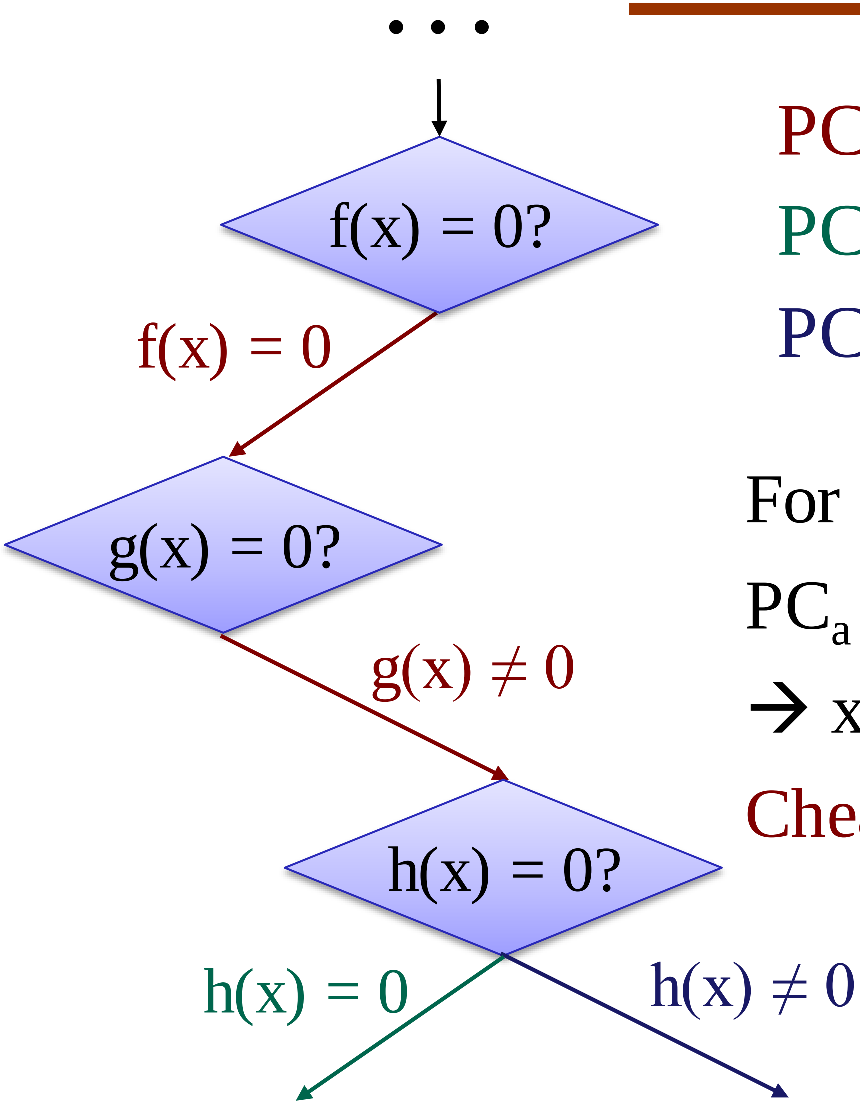
PC_a satisfiable ➔ at least one of PC_b or PC_c satisfiable

PC_b UNSAT ➔ PC_c SAT (valid)

PC_c UNSAT ➔ PC_b SAT (valid)

PC_b SAT ➔ ?

Example optimisation



PC_a: $f(x) = 0 \land g(x) \neq 0$

PC_b: $f(x) = 0 \land g(x) \neq 0 \land h(x) = 0$

PC_c: $f(x) = 0 \land g(x) \neq 0 \land h(x) \neq 0$

For each SAT query, we ask for a CEX!

PC_a SAT with CEX $x = 10$

→ $x = 10$ a solution for either PC_b or PC_c

Cheap to check!

Cex Caching: generalisation

$2 * y < 100$
$x > 3$
$x + y > 10$

$x = 5$
$y = 15$

$2 * y < 100$
$x + y > 10$

Eliminating constraints
cannot invalidate solution

$x = 5$
$y = 15$

$2 * y < 100$
$x > 3$
$x + y > 10$
$x < 10$

Adding constraints often
does not invalidate solution

$x = 5$
$y = 15$

[OSDI'08]

Total queries vs STP queries

Application	Queries/s	Queries	STP queries
[	7.9	30,838	30,613
base64	42.2	184,348	47,600
chmod	12.6	46,438	37,911
comm	305.0	1,019,973	21,720
csplit	63.5	285,655	33,623
dircolors	4,251.7	5,609,093	2,077
echo	4.5	16,318	764
env	26.3	96,425	38,047
factor	22.6	80,975	6,189
join	3,401.2	5,362,587	4,963
ln	24.5	91,812	40,868
mkdir	7.2	26,631	25,622

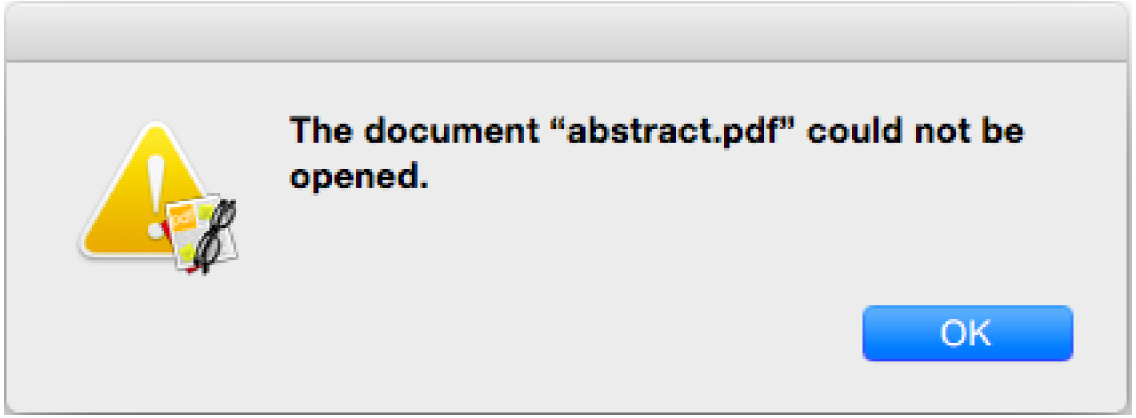
[CAV'13]

Docovery: recovering broken documents

Joint work with:

Tomasz Kuchta, Miguel Castro, Manuel Costa [ASE 2014]

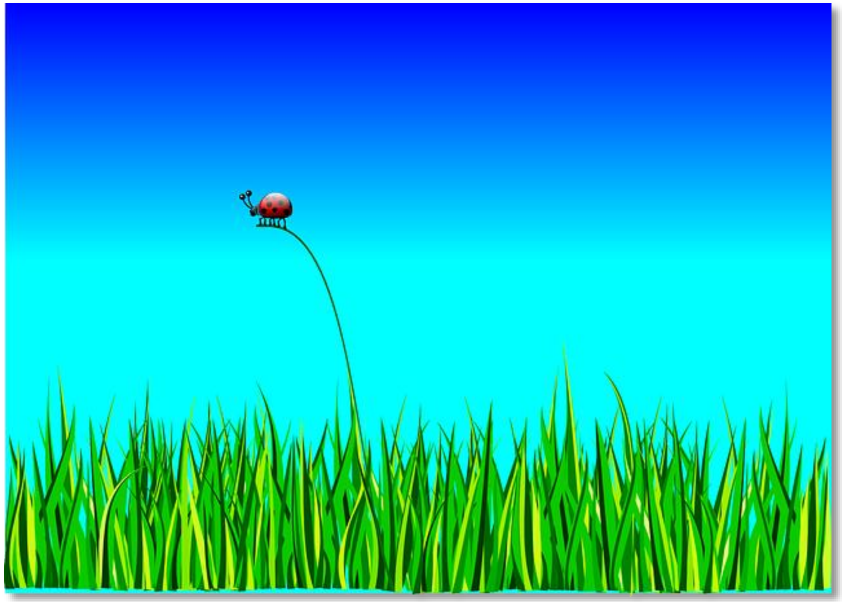
Motivation



Corrupt Documents

Storage failure, network transfer failure, power outage

Application Bugs



Buffer overflows, assertion failures, exceptions

Incompatibility across versions / applications

Research Question

Is it possible to fix a broken document, without assuming any input format, in a way that preserves the original contents as much as possible?

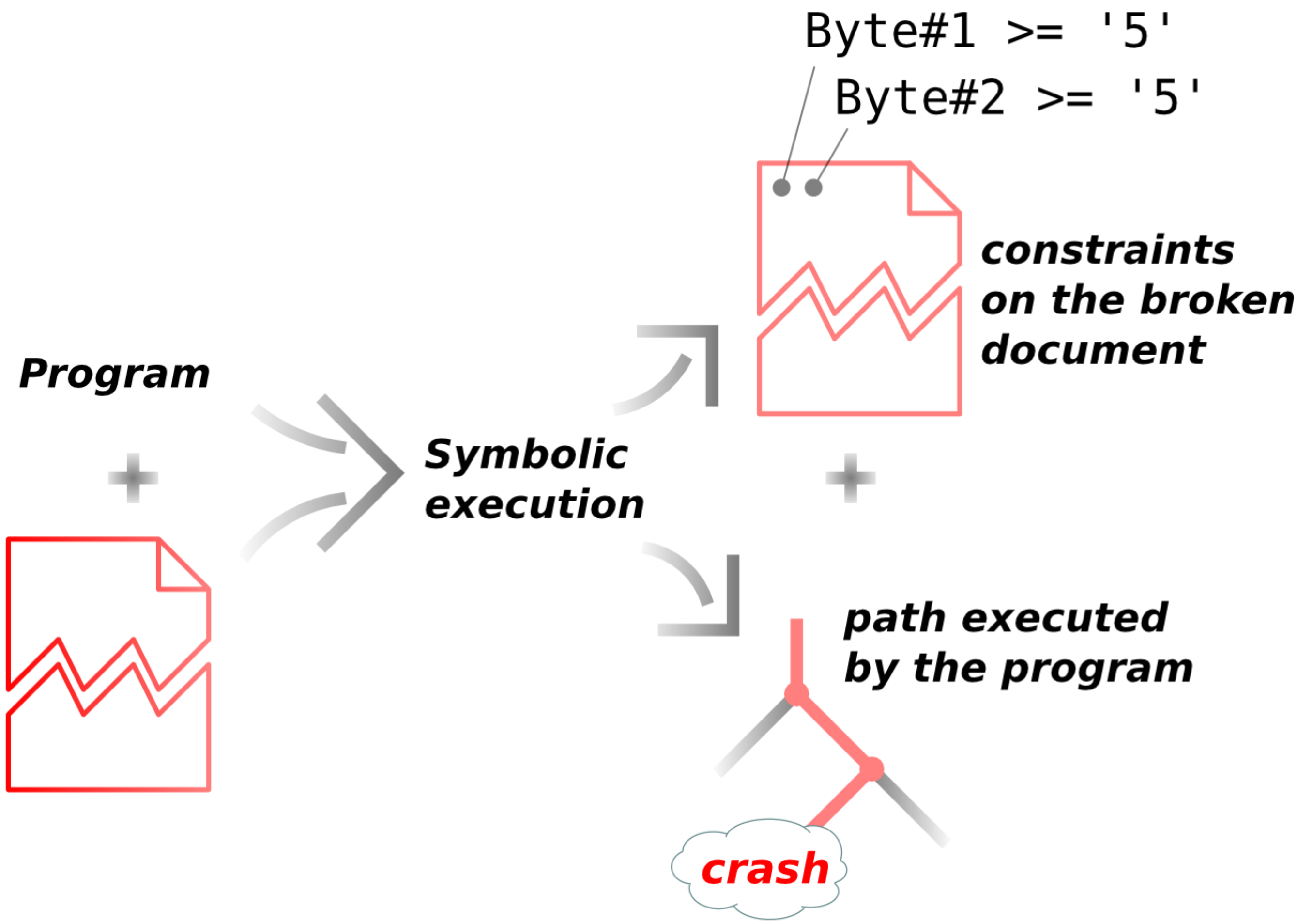
Docovery

Program

+

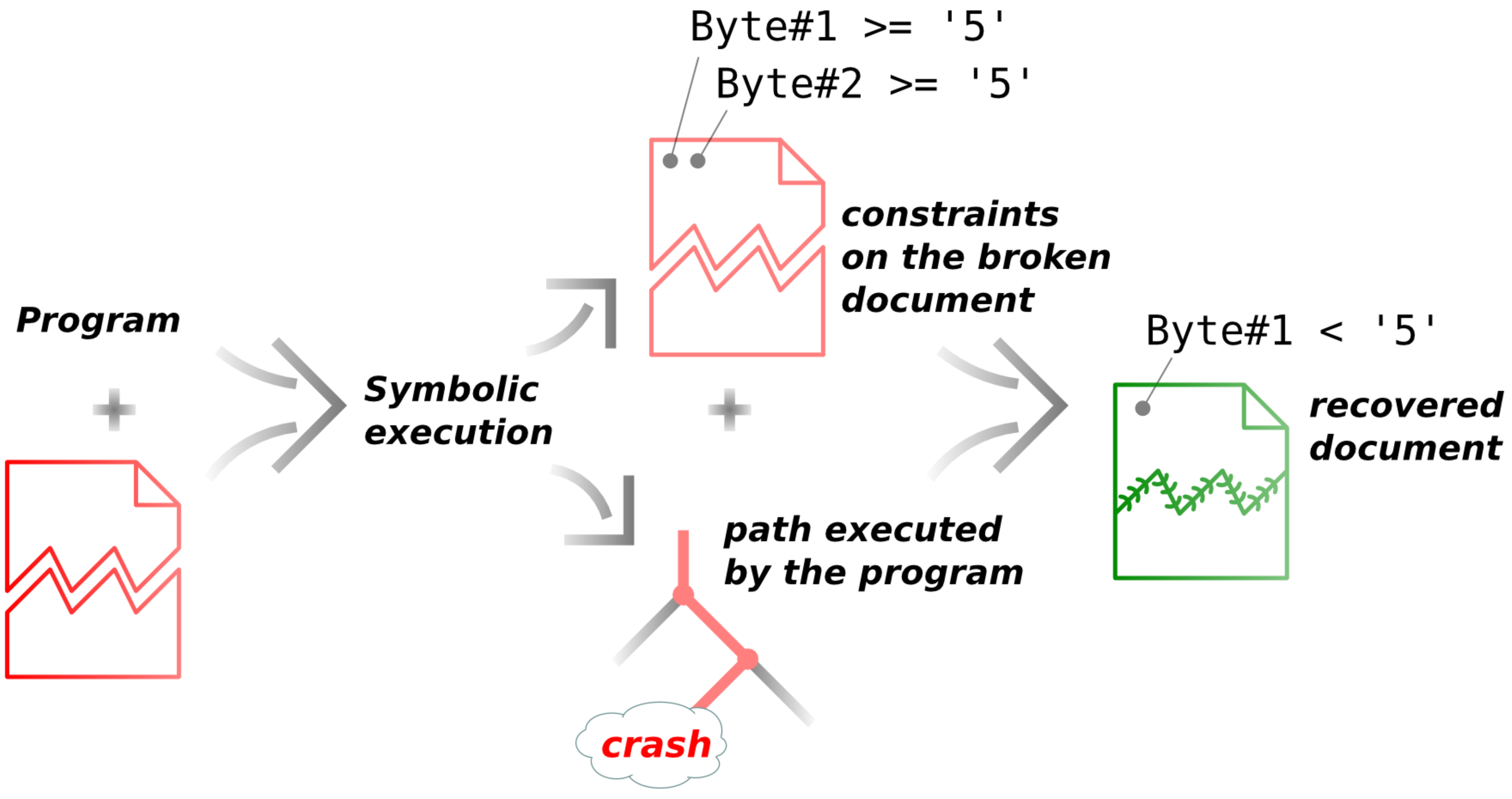
[ASE 2014]

Docovery



[ASE 2014]

Docovery



[ASE 2014]

Constraint Solving Challenges

1) Huge number of constraints
   - we don't choose the input size!

(Partial) solution: initial taint tracking stage to identify problematic bytes

Constraint Solving Challenges

2) Need counterexamples similar to the initial bytes!

- no such mechanism in existing solvers (AFAWK)

Algorithm(PC, bytes b, initial values v)

for each b_K with initial value v_K

if ($b_K = v_K$) is satisfiable (solver call)

then $PC = PC \wedge (b_K = v_K)$

else get new value for b_K from solver

One solver call for each byte… can the solver help?

Initial study on 4 medium-sized apps

pr – a pagination utility

pine – a text-mode e-mail client

dwarfdump – a debug information display tool

readelf – an ELF file information display tool

Benchmark	Document type	Document Sizes
`pr`	Plain text	up to 256 pages / 1080 KB
`pine`	MBOX mailbox	up to 320 e-mails / 2.3 MB
`dwarfdump`	DWARF executables	up to 1.1 MB
`readelf`	ELF object files	up to 1.5 MB

Examined issues caused by application bugs

Known, real-world bugs

pr, pine, readelf – buffer overflow

dwarfdump – division by zero

Benchmark	'Buggy' sequence
pr	Lorem ipsum...**0x08 0x08...0x09** EOF
pine	...From: "**\"\"\"\"\"\"\"\...\"\"\"\"**"@host.fubar...
dwarfdump	...GCC: (Ubuntu/Linaro 4.6.3...**0x00** 0x00...
readelf	...**0xFD 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF**...

Results

Benchmark	Document sizes	Candidates/document/run	Number of changed bytes
`pr`	up to 256 pages / 1080 KB	3	1
`pine`	up to 320 e-mails / 2.3 MB	8 – 27	1 – 24
`dwarfdump`	up to 1.1 MB	2	1
`readelf`	up to 1.5 MB	1 – 3	1 – 8

Number of candidates and changed bytes not influenced by document size

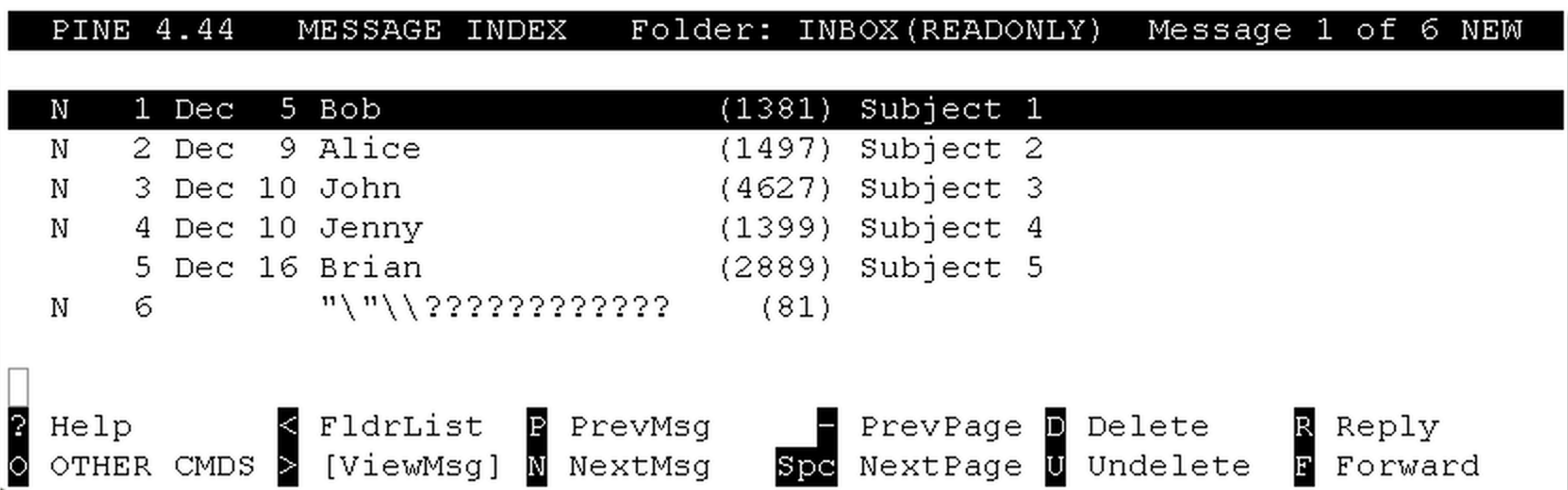
Pr: recovery candidates

Document	'Buggy' sequence
Original	`Lorem ipsum...0x08 0x08...`**`0x09`**` EOF`
Candidate A	`Lorem ipsum...0x08 0x08...`**`0x00`**` EOF`
Candidate B	`Lorem ipsum...0x08 0x08...`**`0x0C`**` EOF`
Candidate C	`Lorem ipsum...0x08 0x08...`**`0x0A`**` EOF`

All the candidates avoid the crash and print the text correctly

Pine: recovery candidates

Document	'Buggy' sequence
Original	`From: "\"\"\"\".................\""@host.fubar`
Candidate A	`From: "\"\...\0x0E...\0x0E\"...\""@host.fubar`
Candidate B	`From: "\"\...\\\0x0E..\0x0E\"..\""@host.fubar`
Candidate C	`From: "\"\...\0x00\"...........\""@host.fubar`



```
PINE 4.44    MESSAGE INDEX    Folder: INBOX(READONLY)   Message 1 of 6 NEW

 N    1 Dec  5 Bob                     (1381) Subject 1
 N    2 Dec  9 Alice                   (1497) Subject 2
 N    3 Dec 10 John                    (4627) Subject 3
 N    4 Dec 10 Jenny                   (1399) Subject 4
      5 Dec 16 Brian                   (2889) Subject 5
 N    6        "\"\\????????????         (81)

? Help        < FldrList   P PrevMsg     - PrevPage D Delete    R Reply
O OTHER CMDS  > [ViewMsg]  N NextMsg   Spc NextPage U Undelete  F Forward
```

All the candidates avoid the crash and display mailbox

Docovery: limitations

- Large documents where taint tracking not that successful
- Highly-structured documents

- *Huge number of possible candidates*
- *Huge constraint sets*

- On-going work to make it scale to PDF docs

Testing and Verifying Optimizations

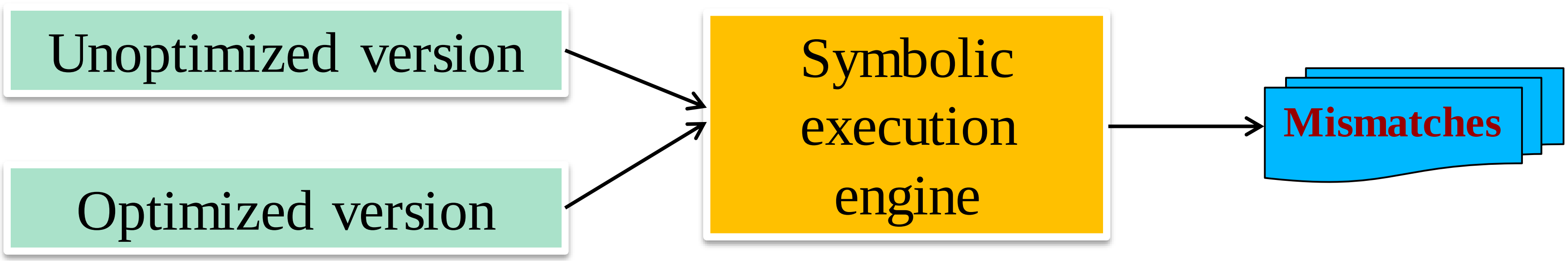
Joint work with:

Peter Collingbourne, Paul Kelly [EuroSys 2011, HVC 2011]

Testing Semantics-Preserving Evolution via Crosschecking

Lots of available opportunities as code is:

<u>Optimized frequently</u> <u>Refactored frequently</u>



We can find any mismatches in their behavior by:

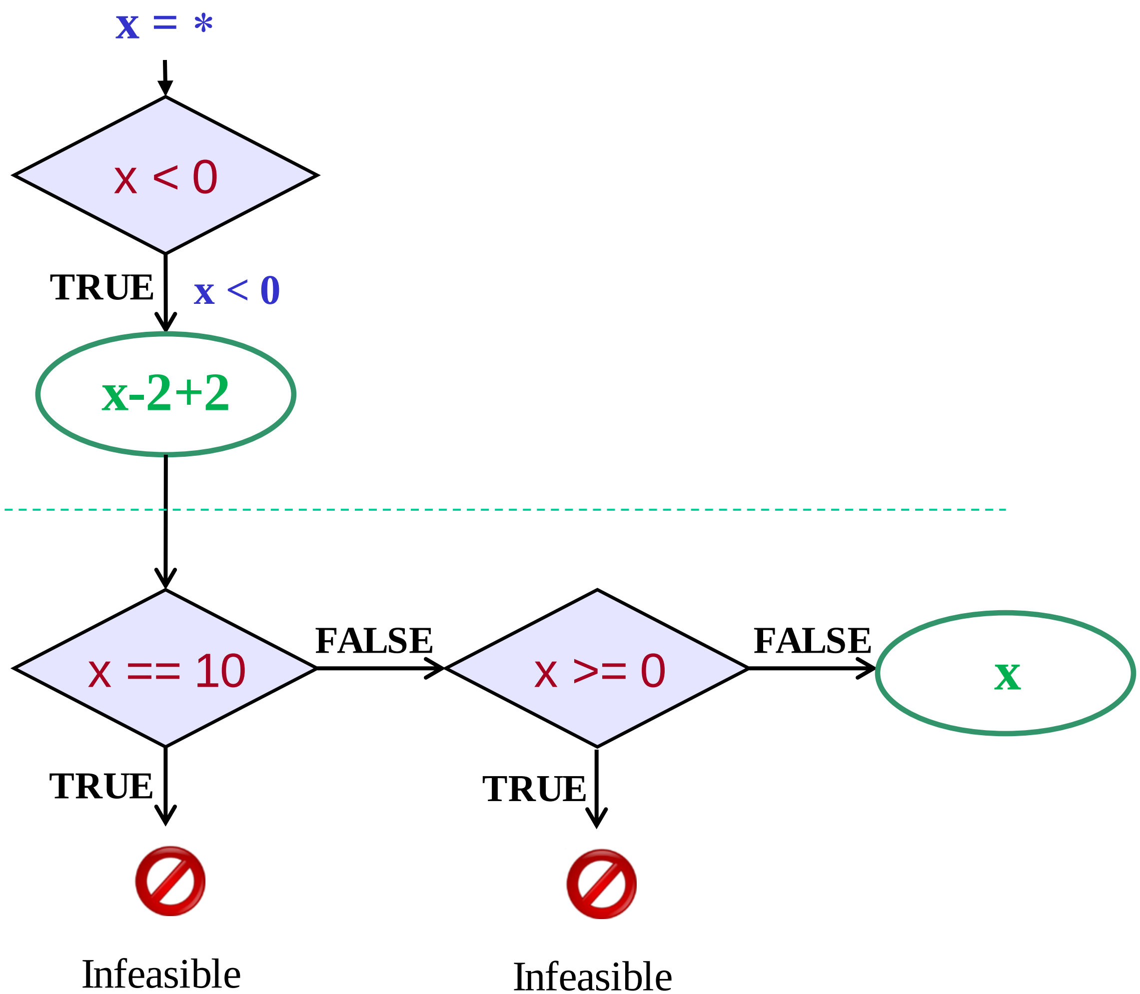
1. Using symbolic execution to explore multiple paths
2. Comparing the (symbolic) output b/w versions

Crosschecking Two Software Versions

```
if (x < 0)
    x -= 2;
else
    if (x%2 != 0)
      x--;
return x+2;
```

```
if (x == 10)
   return 12;

if (x >= 0) {
   if (x%2 == 0)
      x++;
   x++;
}
return x;
```

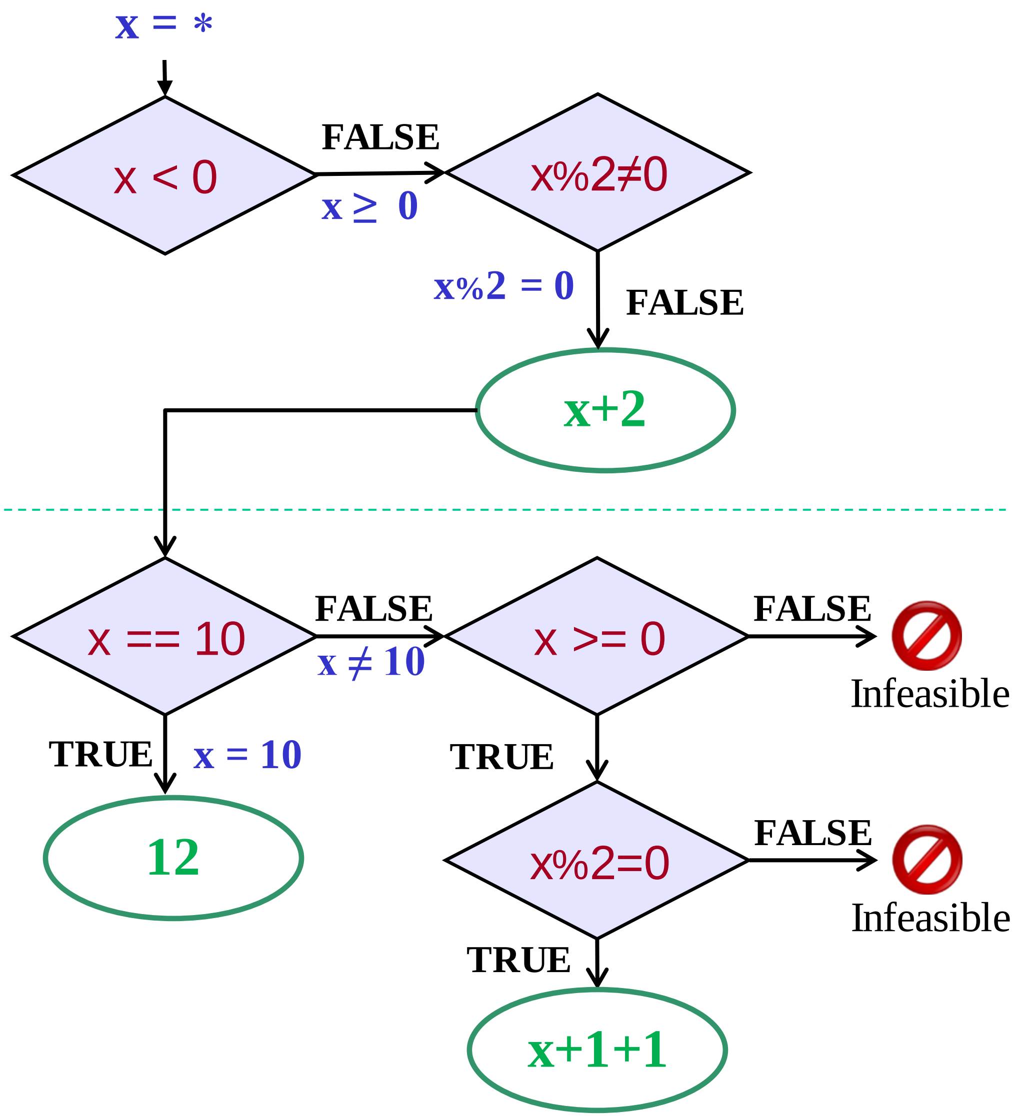


Crosschecking Two Software Versions

```
if (x < 0)
    x -= 2;
else
    if (x%2 != 0)
      x--;
return x+2;
```

```
if (x == 10)
   return 12;

if (x >= 0) {
   if (x%2 == 0)
      x++;
   x++;
}
return x;
```

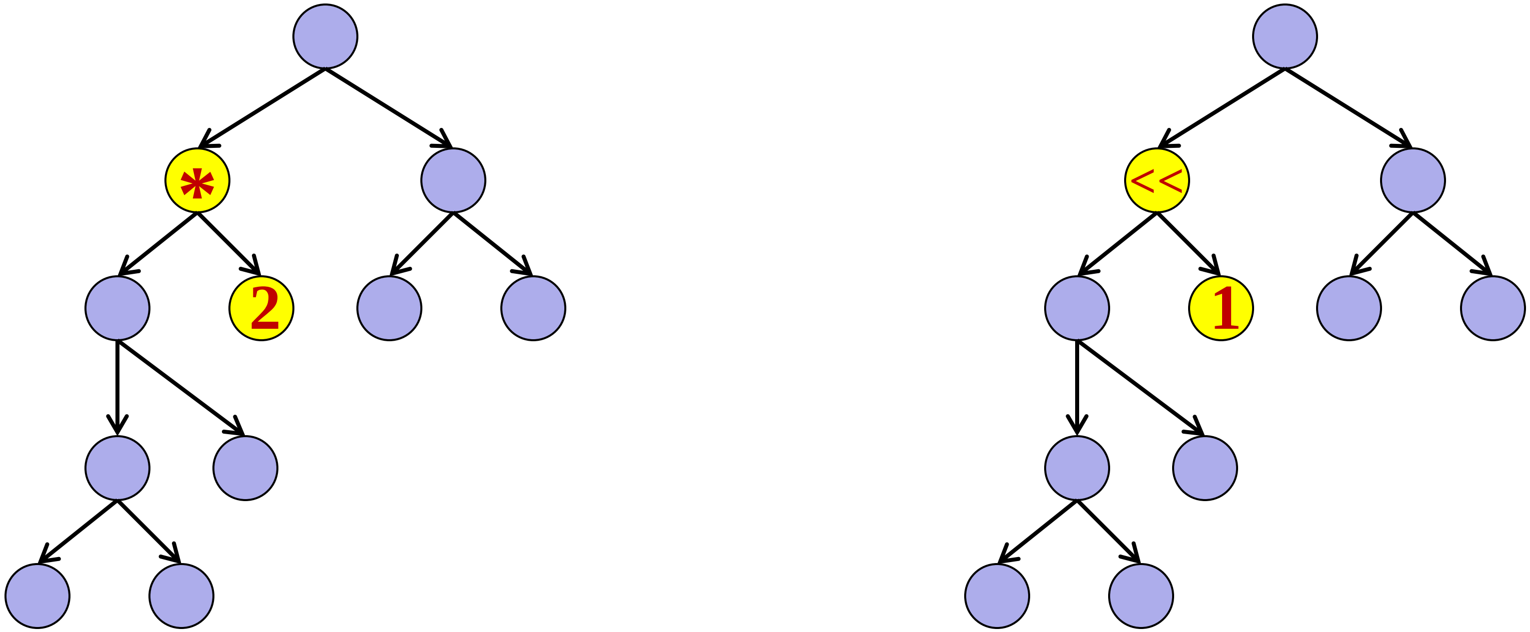


Crosschecking: Discussion

- Can find semantic errors
- No need to write (additional) specifications
- Crosschecking queries can be solved faster
- Can support constraint types not (efficiently) handled by the underlying solver, e.g., floating-point

Many crosschecking queries can be *syntactically* proven to be equivalent

Crosschecking: Advantages



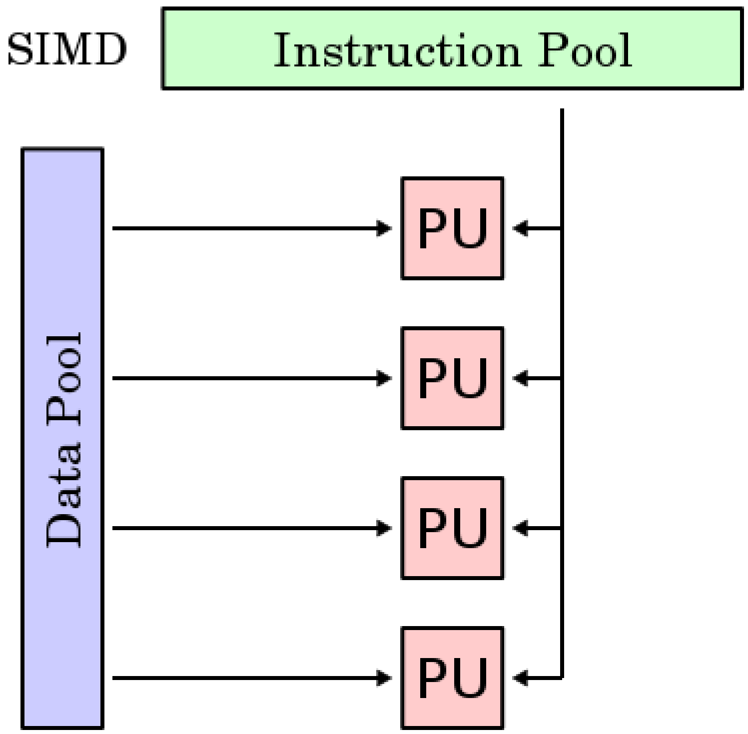
Many crosschecking queries can be *syntactically* proven to be equivalent via simple *rewrite rules*

- Any work on designing constraint solving algorithms for crosschecking queries?

SIMD Optimizations

Most processors offer support for SIMD instructions

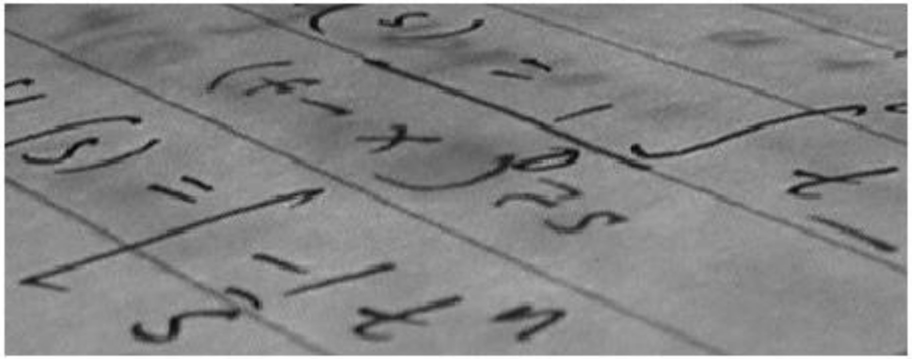
- Can operate on multiple data concurrently
- Many algorithms can make use of them (e.g., computer vision algorithms)



[EuroSys 2011]

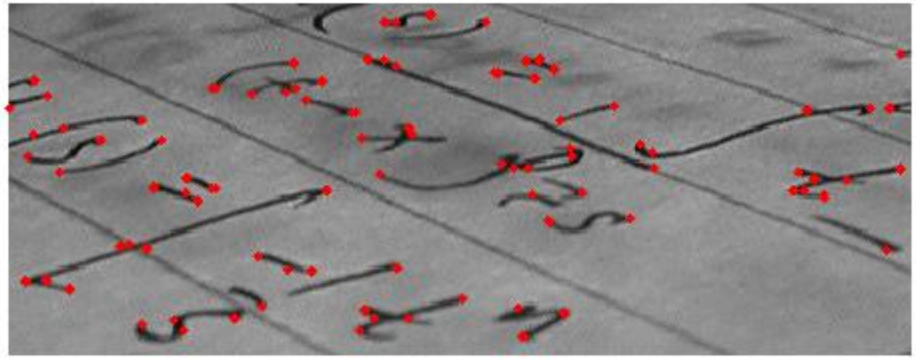
OpenCV

Popular computer vision library from Intel and Willow Garage



[Corner detection algorithm]

Computer vision algorithms were optimized to make use of SIMD



OpenCV Results

- Crosschecked 51 SIMD-optimized versions against their reference scalar implementations
  - Verified the correctness of 41 of them up to a certain image size (*bounded verification*)
- Key idea:
  - Tame path explosion by statically merging paths

[EuroSys 2011]

OpenCV Results

- Crosschecked 51 SIMD-optimized versions against their reference scalar implementations
  - Found mismatches in 10
- Most mismatches due to tricky FP-related issues:
  - Precision
  - Rounding
  - Associativity
  - Distributivity
  - NaN values

[EuroSys 2011]

OpenCV Results

Surprising find: min/max not commutative nor associative!

```
min(a,b) = a < b ? a : b

a < b (ordered) → always returns false if one
                  of the operands is NaN

min(NaN, 5) = 5
min(5, NaN) = NaN

min(min(5, NaN),  100) = min(NaN, 100) = 100
min(5, min(NaN, 100))  = min(5, 100) = 5
```

GPGPU Optimizations

Scalar vs. GPGPU code

[HVC 2011]

Constraint Solving in Symbolic Execution

- Constraint solving plays a key role in symbolic execution
- Important to take advantage of the characteristics of the queries generated during symbolic execution
  - Bug-finding in low-level systems and security-critical code: *need to solve lots of sat and cex queries fast*
  - Recovery of broken documents: *need to generate counterexamples similar to the original bytes*
  - Testing and bounded verification of optimisations: *many queries can be solved fast via simple syntactic rewrite rules*