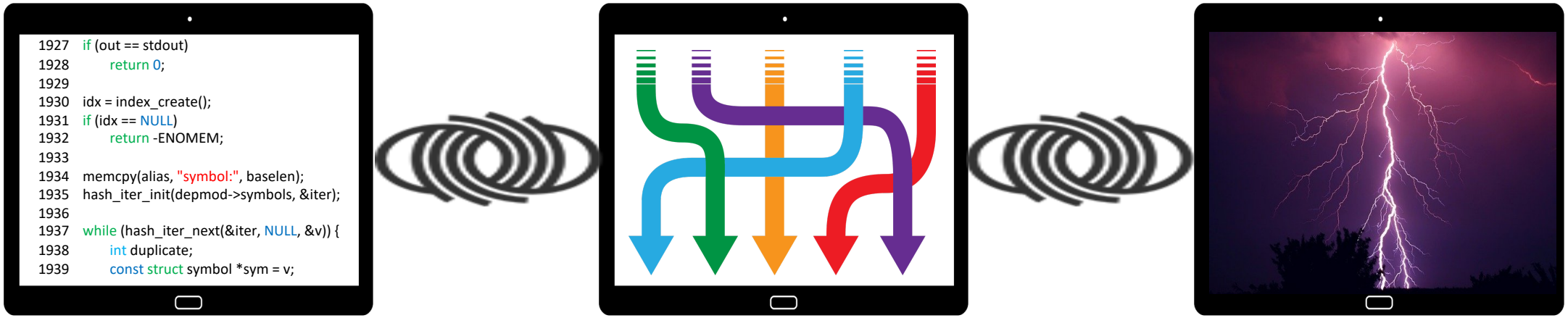
Dynamic Symbolic Execution for Software Analysis

Cristian Cadar
Department of Computing
Imperial College London

Imperial College London

GTMFS
30 January 2020
Sorbonne Université, Paris

Dynamic Symbolic Execution

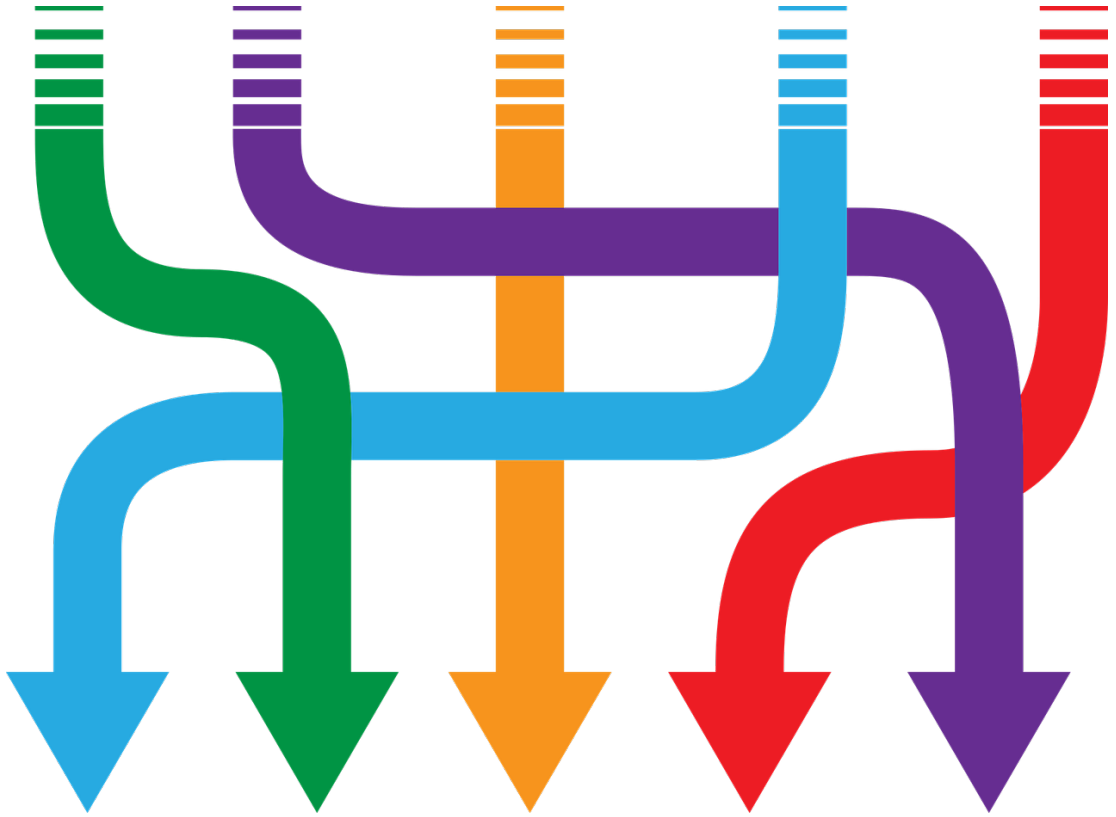


Dynamic Symbolic Execution

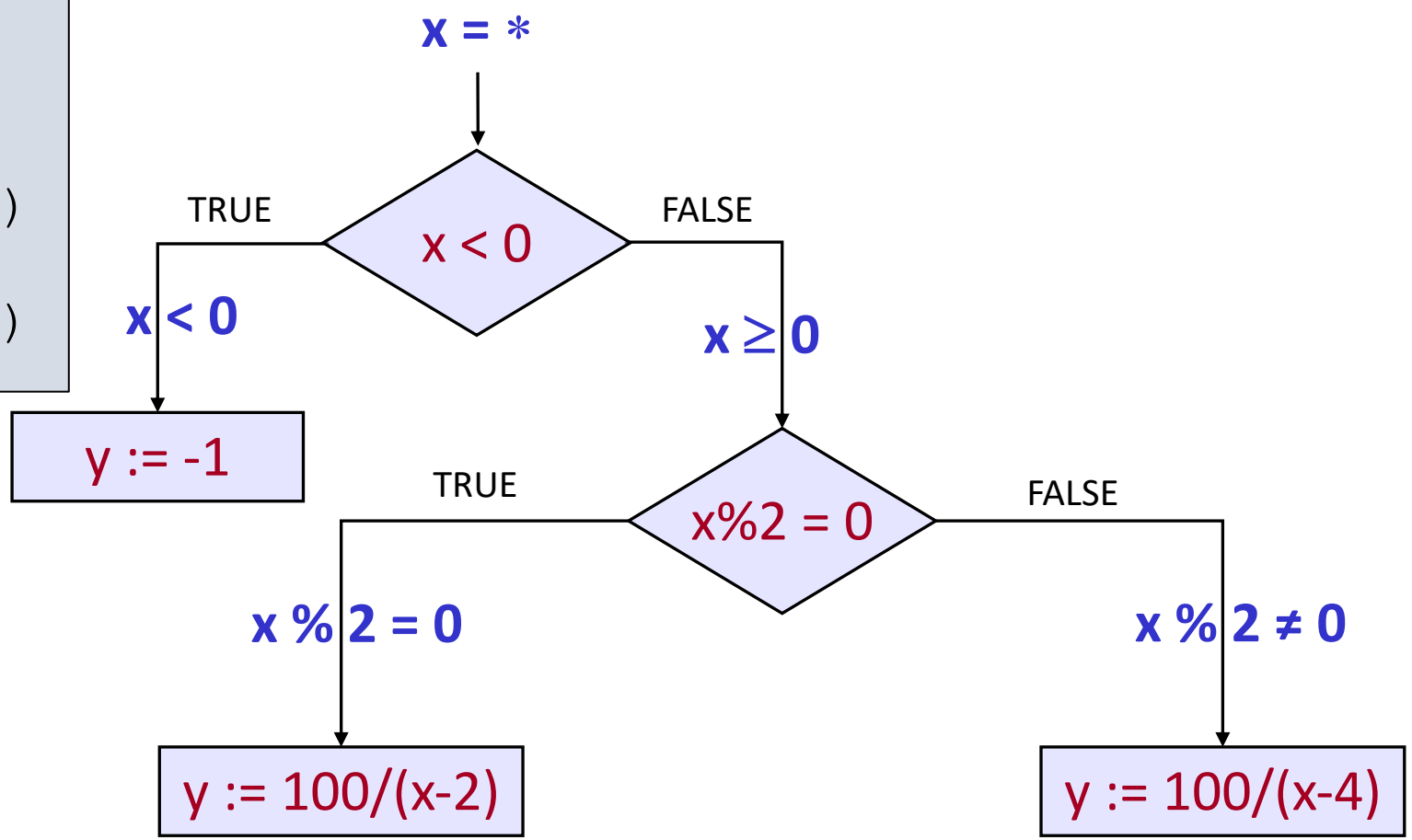
Technique for automatically exploring multiple paths, reasoning about path feasibility using a constraint solver.

Applications in:

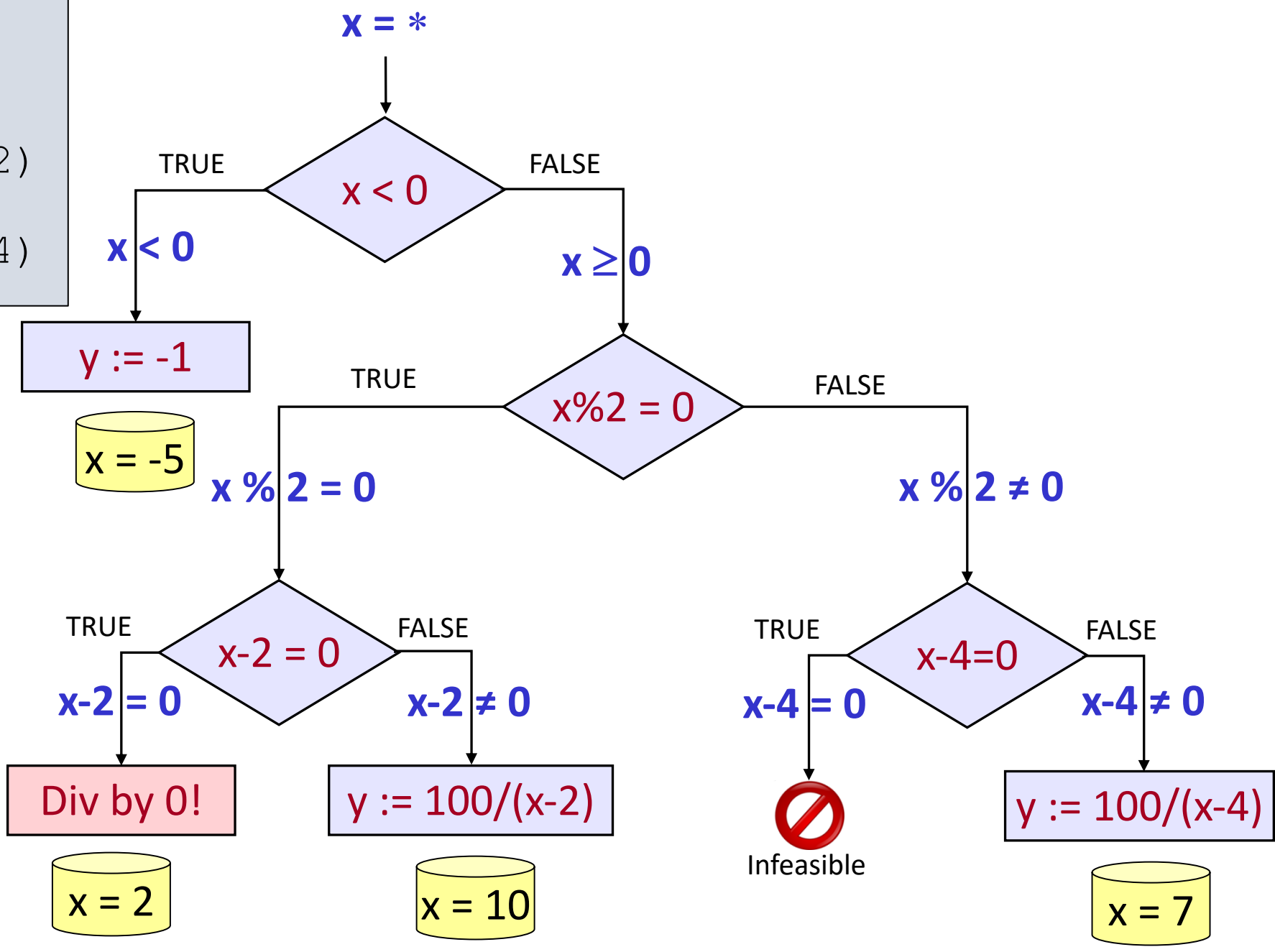
- Bug finding
- Test generation
- Vulnerability detection and exploitation
- Equivalence checking
- Debugging
- Program repair
- Side-chanel analysis
- Refactoring
- etc. etc.



```
if x < 0
   y := -1
else
   if x % 2 = 0
      y := 100 / (x-2)
   else
      y := 100 / (x-4)
```



```
if x < 0
   y := -1
else
   if x % 2 = 0
      y := 100 / (x-2)
   else
      y := 100 / (x-4)
```

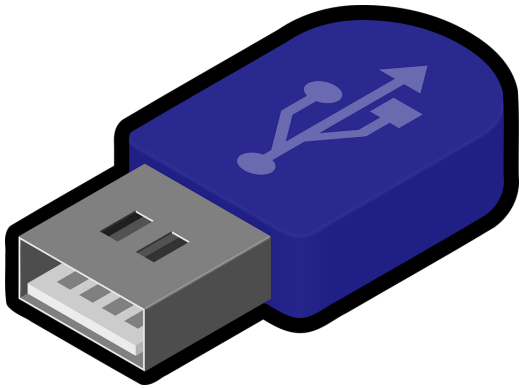


Dynamic Symbolic Execution

- Powerful technique in-between static and dynamic analysis
  - Can drive exploration along many different paths
  - Can reason about all possible values on each explored execution path
  - Can interact with the outside environment
- Can reason about highly complex properties
  - Each path is encoded as a mathematical formula
- Generates concrete test cases for explored paths
  - In particular actual inputs that trigger any discovered bugs

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**

[Joint work with Yang, Sar, Twohey, Engler, IEEE S&P 2008]

Packet of Death: Bonjour

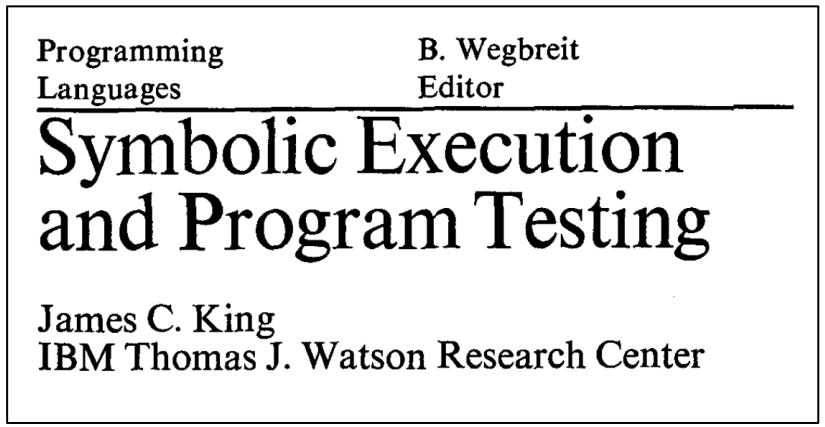
Offset	Hex Values							
0000	0000	0000	0000	0000	0000	0000	0000	0000
0010	003E	0000	4000	FF11	1BB2	7F00	0001	E000
0020	00FB	0000	14E9	002A	0000	0000	0000	0001
0030	0000	0000	0000	055F	6461	6170	045F	7463
0040	7005	6C6F	6361	6C00	000C	0001		

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

[Joint work with Song and Pietzuch, IEEE TSE 2014]

A Bit of History: 1975-76

- Symbolic execution developed independently by several researchers

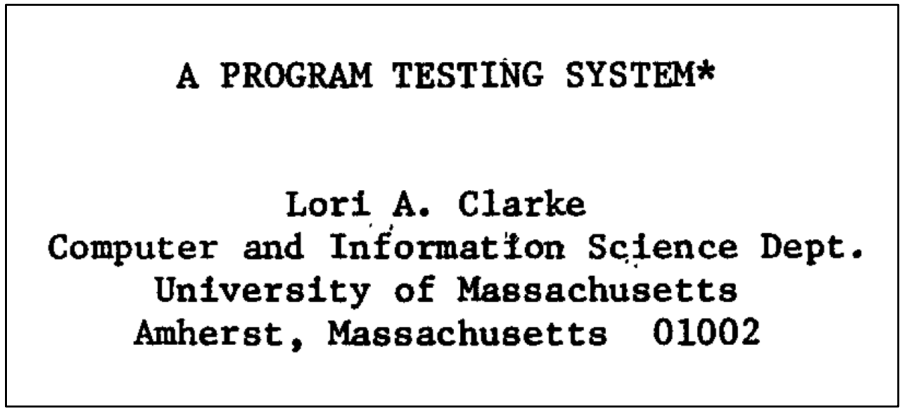


Programming Languages — B. Wegbreit Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

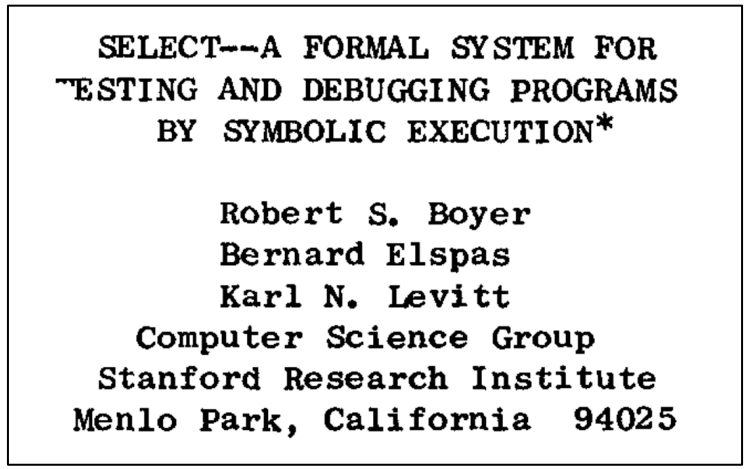
Symbolic execution of PL/I programs



A PROGRAM TESTING SYSTEM*

Lori A. Clarke
Computer and Information Science Dept.
University of Massachusetts
Amherst, Massachusetts 01002

Symbolic execution of Fortran programs



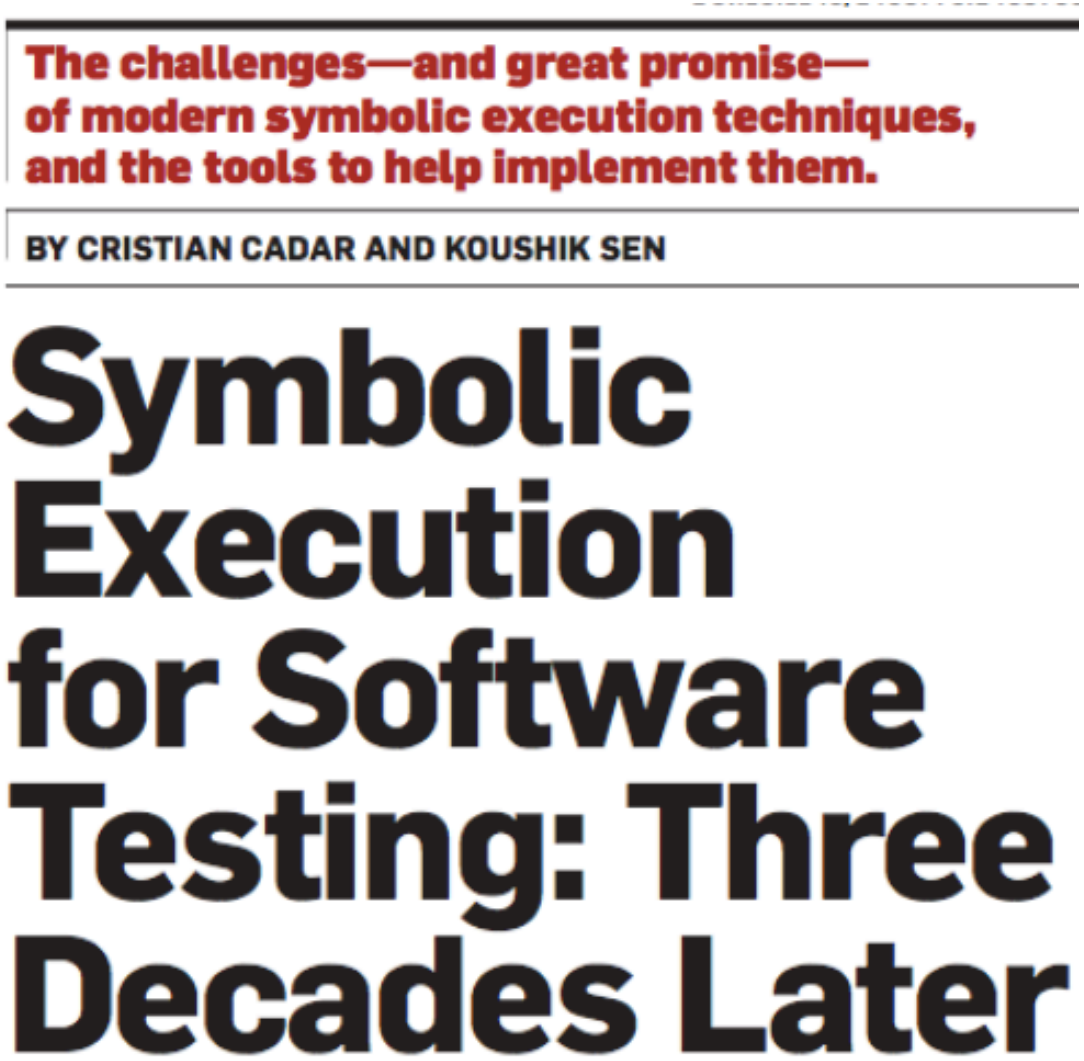
SELECT--A FORMAL SYSTEM FOR TESTING AND DEBUGGING PROGRAMS BY SYMBOLIC EXECUTION*

Robert S. Boyer
Bernard Elspas
Karl N. Levitt
Computer Science Group
Stanford Research Institute
Menlo Park, California 94025

Symbolic execution of LISP programs

1975-76

A Bit of History



The challenges—and great promise—
of modern symbolic execution techniques,
and the tools to help implement them.

BY CRISTIAN CADAR AND KOUSHIK SEN

Symbolic Execution for Software Testing: Three Decades Later

A Bit of History

- Mixed concrete-symbolic execution

- Availability of precise runtime information
- Ability to interact with the environment
- Partly deal with limitations of solvers
- Only relevant code executed symbolically

DART: Directed Automated Random Testing

Patrice Godefroid Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

DART system for C

Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

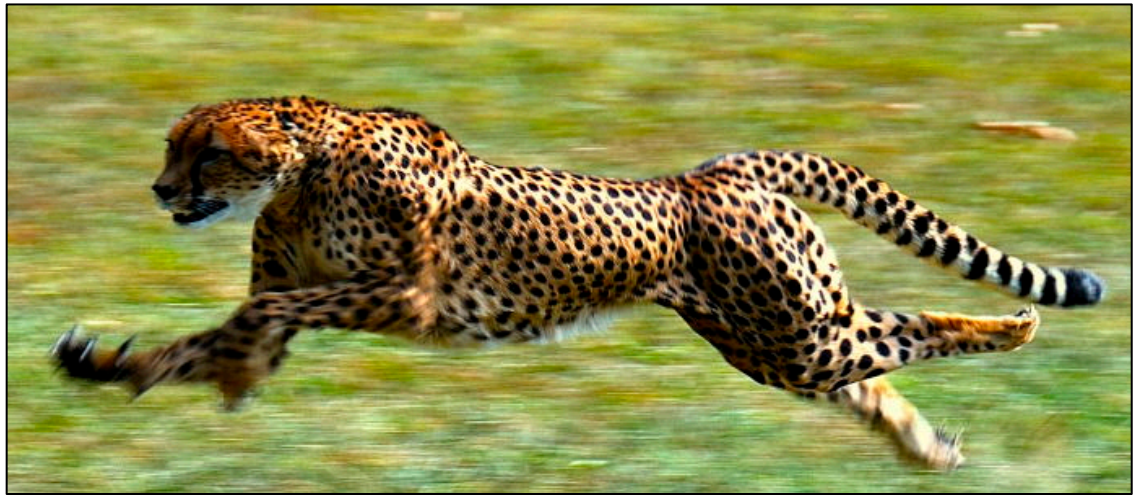
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

EGT system for C

2005

A Bit of History

- Mixed concrete-symbolic execution
- SAT & SMT solvers come of age



DART: Directed Automated Random Testing

Patrice Godefroid Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

2005

A Bit of History

- Mixed concrete-symbolic execution
- SAT & SMT solvers come of age
- Focus on bug-finding

DART: Directed Automated Random Testing

Patrice Godefroid Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

2005

Fifteen Years Later

KLOVER

Miasm

PyExZ3

SymDroid

PathGrind

jCUTE

Otter

SAGE

CREST

SymJS

BinSE

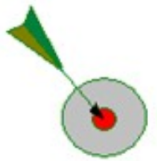
Jalangi2

BINSEC

CUTE

Symbolic PathFinder

angr



DART

Kite

LDSE

Pex

Rubyx

JDart

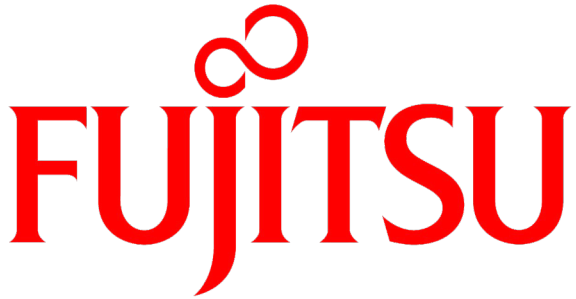
S²E

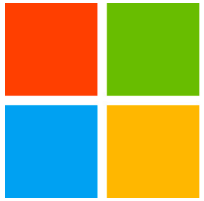
CATG

CiVL

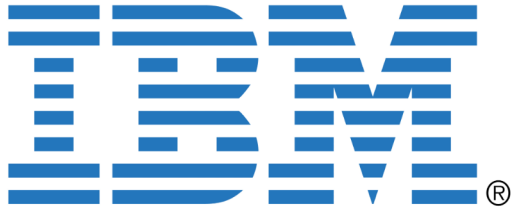
Mayhem

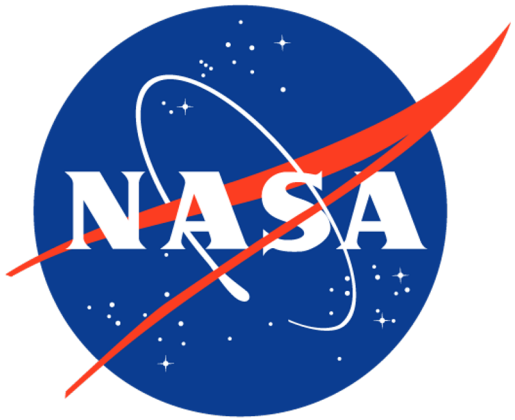
Outside Academia





Microsoft





DARPA Cyber Grand Challenge (2015-16)

"It is a race to find, diagnose, and fix software flaws in real time in an adversarial environment."

Webpage: **klee.github.io**

Code: **https://github.com/klee**

Web version: **http://klee.doc.ic.ac.uk**

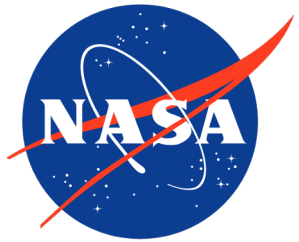
- Symbolic execution for LLVM bitcode, primarily targeting C
- Started at Stanford [Cadar, Dunbar, Engler, OSDI 2008], incorporating the lessons from our prior system EXE [Cadar, Ganesh, Pawlowski, Engler, Dill, CCS 2016]
- Evolved to incorporate many of the ideas we developed in the last decade
- Open source, available on GitHub (**1000+ stars, 330+ forks, 50+ contributors**)
- Widely used in academia and industry

KLEE – Academic Impact

- **ACM CCS Test of Time Award 2016** for EXE paper, which laid the foundations of KLEE
- **ACM SIGOPS Hall of Fame Award 2018** for KLEE paper
- Over **2,600 citations** to KLEE paper (Google Scholar, Jan. 2020)
- From **many different fields** (in no particular order):
  - Testing: ISSTA, ICST, etc.
  - Verification: CAV, HVC, etc.
  - Systems: OSDI, SOSP, EuroSys, etc.
  - Software engineering: ICSE, FSE, ASE, etc.
  - Security: CCS, IEEE S&P, USENIX Security, etc.
  - etc.

KLEE – Impact Outside Academia

Many known uses or trials:

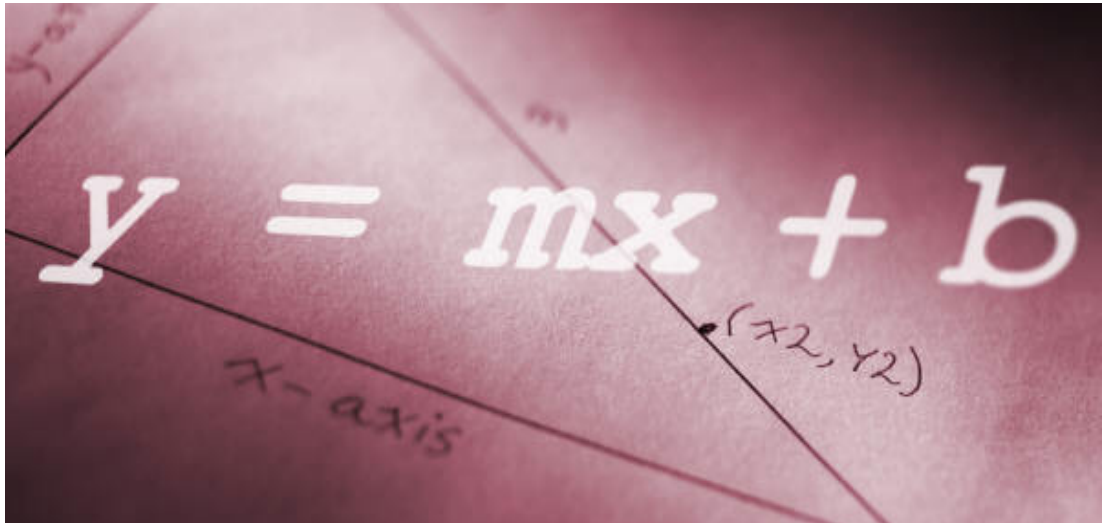


Some publications/talks/blog posts:

- Fujitsu: [PPoPP 2012], [CAV 2013], [ICST 2015], [IEEE Software 2017], [KLEE Workshop 2018]
- Hitachi: [CPSNA 2014], [ISPA 2015], [EUC 2016]
- Trail of Bits: https://blog.trailofbits.com/
- Intel: [WOOT 2015]
- NASA Ames: [NFM 2014]
- Baidu: [KLEE Workshop 2018, IEEE S&P 2020]
- Samsung: [KLEE Workshop 2018]

Ongoing Challenges

Path Explosion



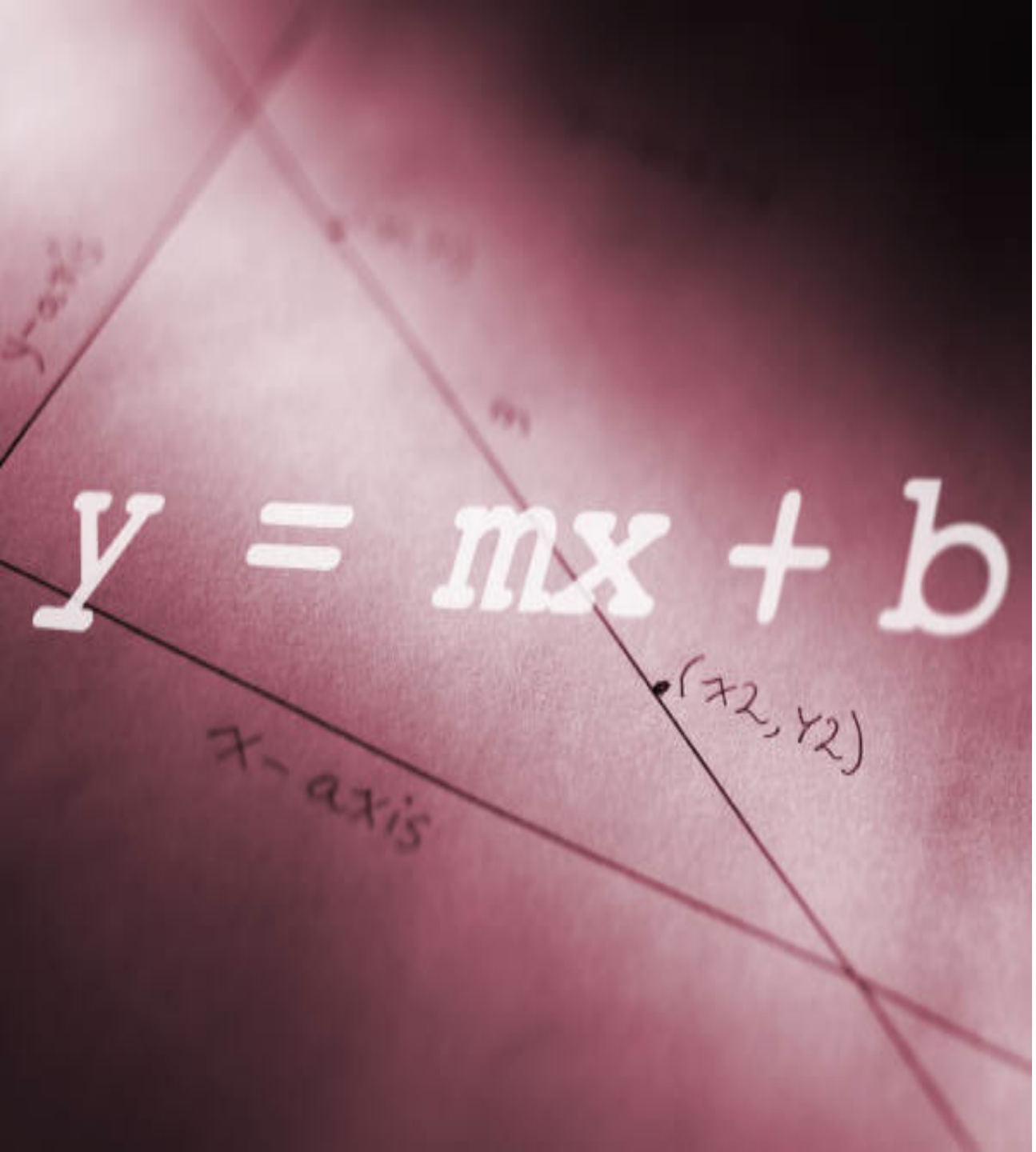
Constraint Solving

Path Explosion

- Real-programs have a huge number of paths
- Exploring the most important first is essential

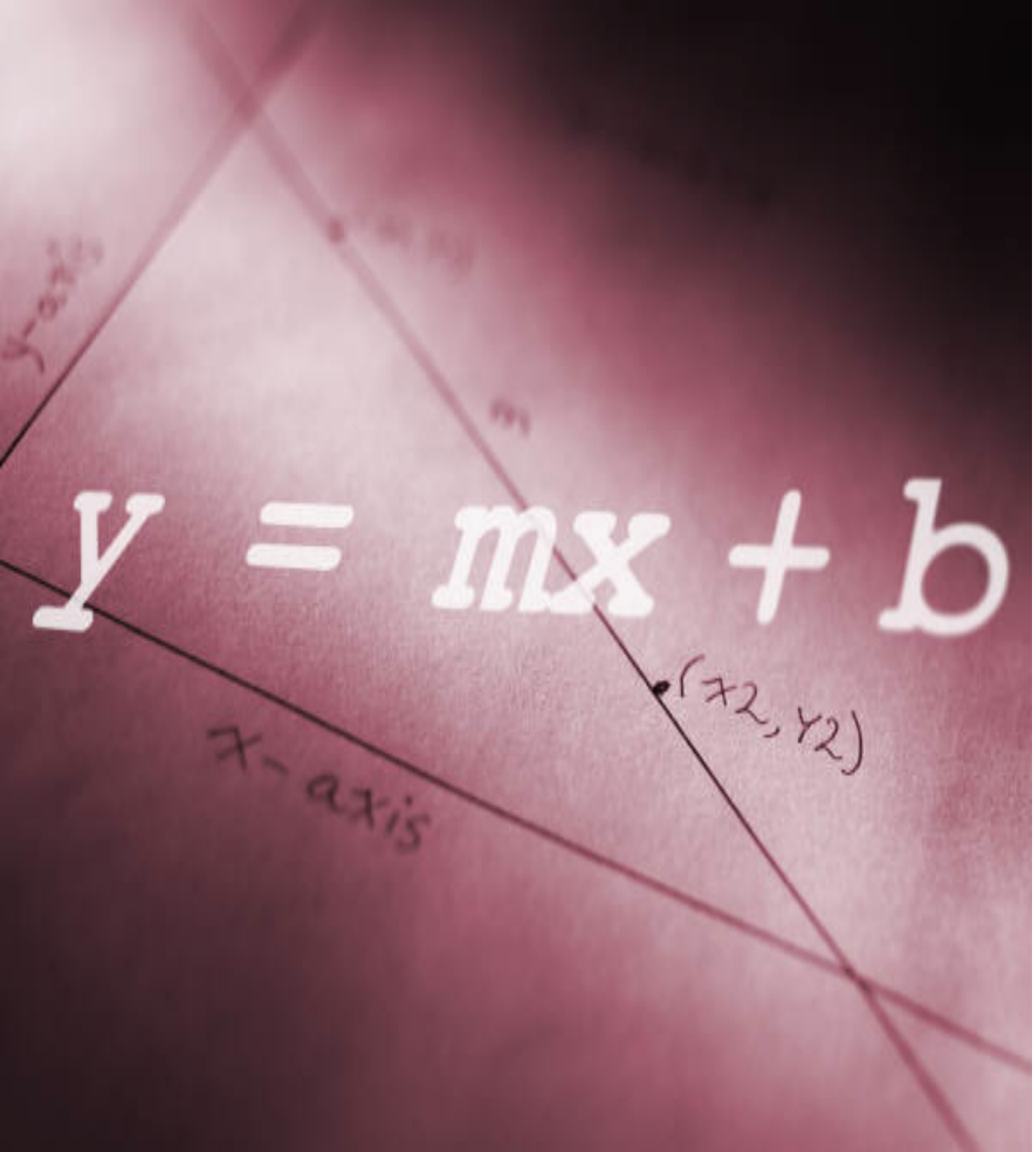
Path Explosion

- Search heuristics
- Eliminating redundant paths
- Code (function, loop, etc.) summaries
- Statically/dynamically merging paths
- Under-constrained execution
- Abstraction-refinement
- Speculatively skipping code fragments
- Using existing test suites to prioritize execution
- Combinations with fuzzing
- Combinations with static analysis



Constraint Solving

- Real programs generate complex queries
- Queries performed at every branch
- To be effective, SymEx needs to solve lots of queries, fast



Constraint Solving

- Simplifying and normalizing constraints
- Eliminating irrelevant constraints
- Caching constraints
- Exploiting subset/superset relations
- Accelerating array constraints
- Performing interval analysis
- Using a portfolio of solvers
- Summarizing string loops
- etc.

Program Transformations

- Key observation: program transformations, such as compiler optimizations, sometimes ***slow down*** symbolic execution analysis!

[Cadar, ESEC/FSE NI 2015]

Targeted Transformations

Unoptimized

```
int get_value(int k){
    return k * k * k;
}

// precond: 0 ≤ k < 1000
int foo(unsigned k) {
    if (get_value(k) > 100000 ||
        get_value(k-1) > 100000)
       return 0;
    else return 1;
}
```

Optimized

```
int values[1000] = {0, 1, 8,
27, 64, 125, 216, 343, 512,
729, 1000, 1331, 1728, ... };

// precond: 0 ≤ k < 1000
int foo(unsigned k) {
    if (values[k] > 100000 ||
        values[k-1] > 100000)
       return 0;
    else return 1;
```

0.2s vs. 50s

250x slower!

$k^3 > 100{,}000$

vs.

$values[k] > 100{,}000 \wedge$
$values[0] = 0 \wedge$
$values[1] = 1 \wedge$
$values[2] = 8 \wedge$
$...$

Index-based Array Transformation

```
char b[256] = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  -1, -1,
-1, -1, -1, -1, -1, - 1, -1, -1, -1, -1, -1, -1, -1,-1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, 62, -1, -1, -1, 63, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, -1, -1, -1, -1, -1, -1, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, -1, -1, -1, -1, -1, -1,
26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1,   -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
};
```

`b[k] > 0`

`k = 43`

`47 <= k <= 57`

`65 <= k <= 90`

`97 <= k <= 122`

[Joint work with Perry, Mattavelli and Zhang, ISSTA 2017]

Value-based Array Transformation

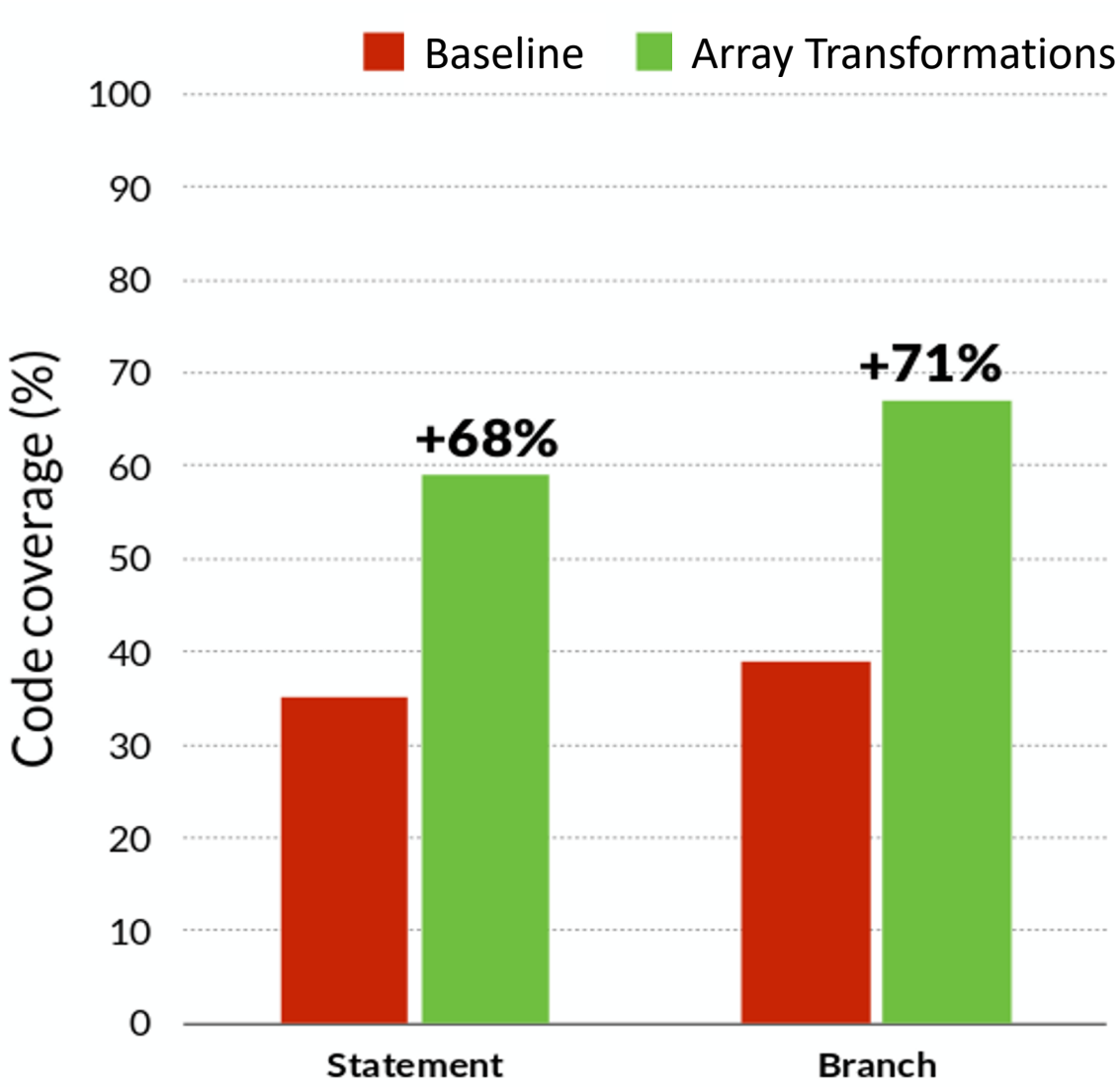
```
char b[256] = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  -1, -1,
-1, -1, -1, -1, -1, - 1, -1, -1, -1, -1, -1, -1, -1,-1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, 62, -1, -1, -1, 63, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, -1, -1, -1, -1, -1, -1, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, -1, -1, -1, -1, -1, -1,
26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1,   -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
};
```

`b[k] > x`

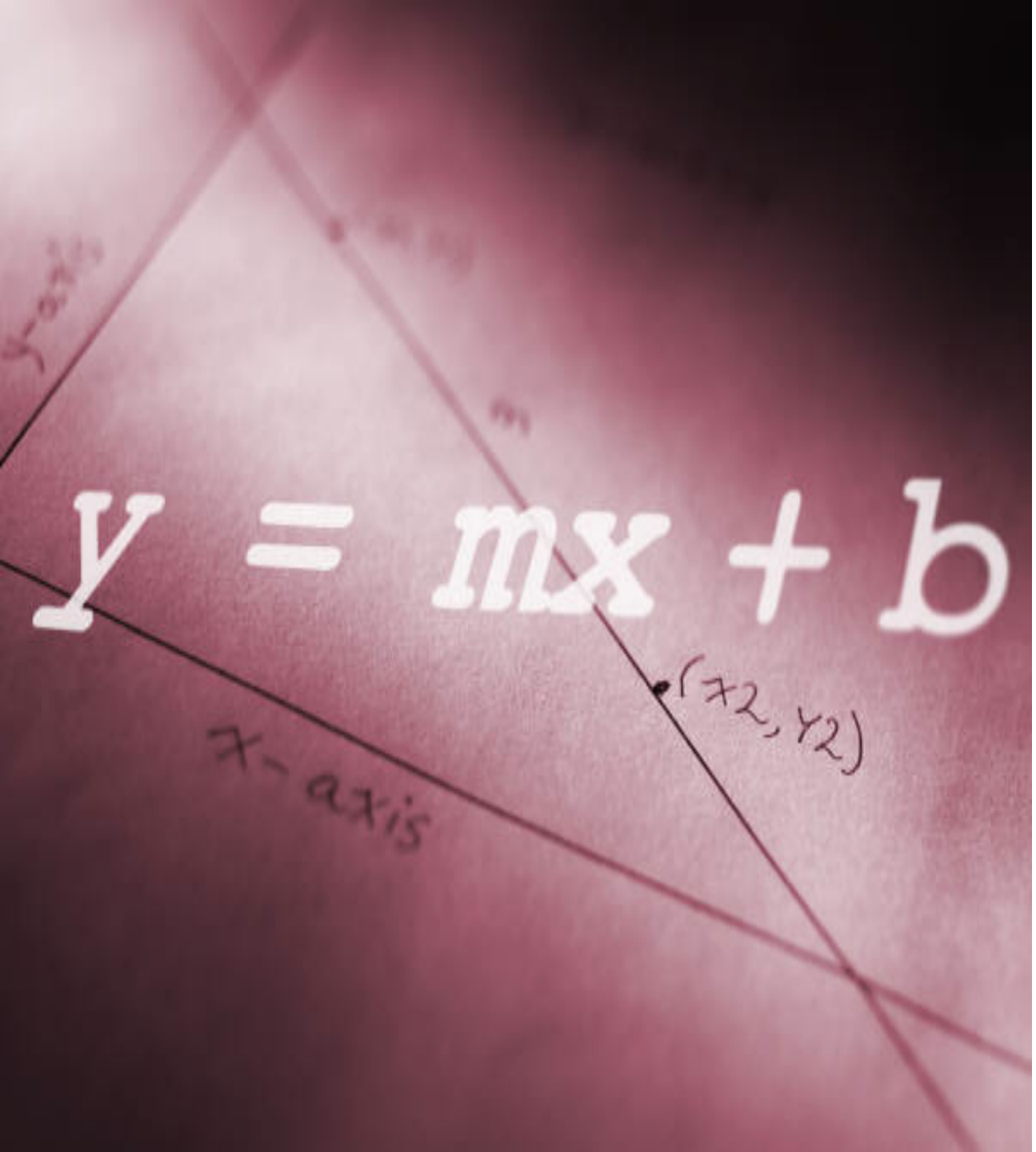
```
ITE(0 ≤ k ≤ 42 ∨ 44 ≤ k ≤ 46 ∨…, -1,
    ITE(k = 43, 62,
        ITE(k = 47, 63,
                    …)))) > x
```

[Joint work with Mattavelli, Perry and Zang, ISSTA 2017]

BC calculator: 6h runs



[Joint work with Mattavelli, Perry, Zhang, ISSTA 2017]



Constraint Solving

- Simplifying and normalizing constraints
- Eliminating irrelevant constraints
- Caching constraints
- Exploiting subset/superset relations
- Accelerating array constraints
- Performing interval analysis
- Using a portfolio of solvers
- Summarizing string loops
- etc.

Summarizing String Loops: Motivation

- Strings everywhere!
- Lots of work on building string constraint solvers from the SMT community
  - E.g., Z3, CVC4, HAMPI
- Let's use them for symbolic execution!

[Joint work with Kapus, Ish-Shalom, Itzhaky, Rinetzky, PLDI 2019]

Problem

- Developers often use custom loops instead of string functions

```
#define whitespace(c) (((c) == '␣') || ((c) == '\t'))
char *p;
for (p = line; p && *p && whitespace (*p); p++)
  ;
```

```
while (*s != '\n')
    s++;
```

```
char *p = path + strlen (path);
for (; *p != '/' && p != path; p--)
  ;
```

```
while (('␣' == *pbeg) || ('\r' == *pbeg)
   || ('\n' == *pbeg) || ('\t' == *pbeg))
   pbeg++;
```

Objective

- Replace custom loops with sequence of primitive pointer operations and calls to standard string functions

```
char *p = line + strspn(line, "␣\t")
```

```
s = rawmemchr(s, '\n');
```

```
pbeg += strspn(pbeg, "␣\r\n\t");
```

```
p = strrchr(path, '/');
p = p == NULL ? path : p;
```

How?

- Counterexample-guided inductive synthesis (based on symex)
- Proof of bounded equivalence, up to a certain string length (based on symex)
- Mathematical proof of unbounded equivalence

[Joint work with Kapus, Ish-Shalom, Itzhaky, Rinetzky, PLDI 2019]

Scope: Memoryless Loops

- Loops conforming to an interface:
  - Argument: single pointer to a string
  - Returns: pointer to an offset in the string
- Only reads the character under current pointer

```
char* loopSummary(char*);
```

Vocabulary for summarizing string loops

string.h functions

- `strspn`
- `strcspn`
- `memchr`
- `strchr`
- `strrchr`
- `strpbrk`

pointer manipulation

- `increment`
- `set to start`
- `set to end`

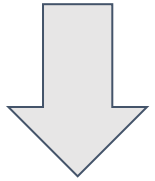
special

- `backward traverse`
- `return`

conditionals

- `is null`
- `is start`

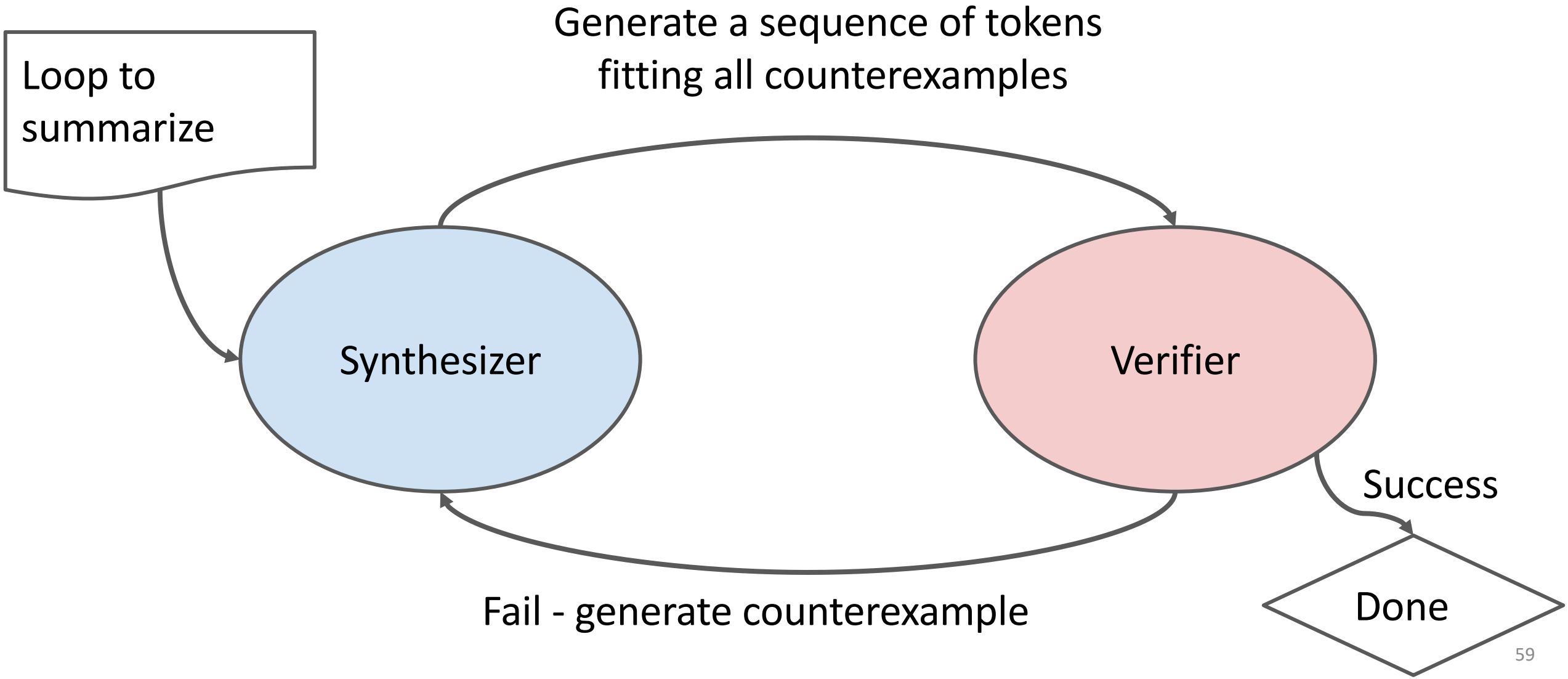
```
char *p;
for (p = line; p && *p && whitespace (*p); p++)
  ;
```



```
char *p = line + strspn(line, "␣\t")
```

Loop summary!

Counter-example guided synthesis



Synthesizer

- Symbolic execution
- Symbolic input: sequence of tokens
- Constrain it to be equivalent on current string (counter)examples
- Ask an SMT solver for a solution

Verifier

- Symbolic execution
  - Bounded equivalence checking strings of length ≤ 3

Mathematical proof

- For memoryless loops:
  - checking lengths ≤ 3 sufficient to show equivalence for any length (proof in the paper)
  - Intuitively the proof depends on the fact that each iteration is independent from previous ones

Synthesis Evaluation

- 13 open-source programs
- Extracted 115 memoryless loops
- 88/115 successfully synthesized within 2h*
- 81 within 5 minutes

*Gaussian process optimization to optimize the vocabulary

patch

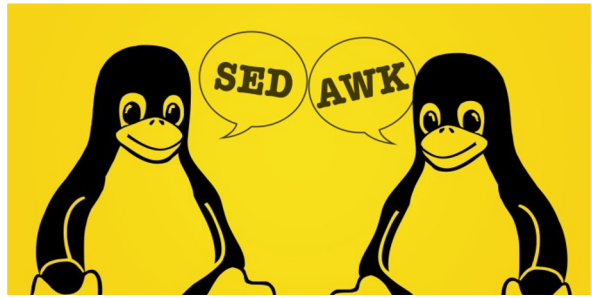
libosip

git

diff

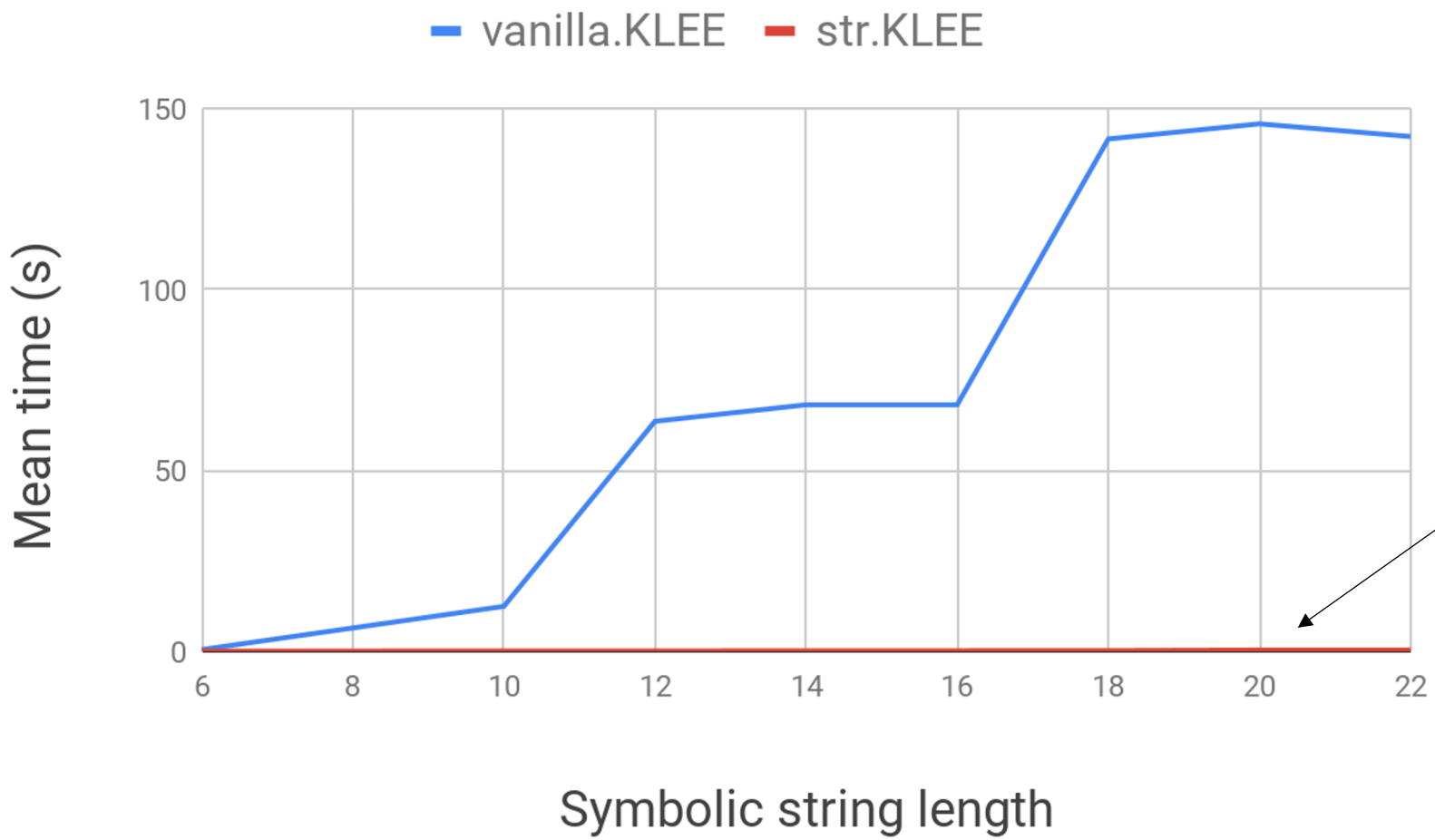
m4

make



Impact of string solvers (KLEE+Z3str) on Sym Ex

Average across loops, 2min timeout



Can reason about unbounded string lengths

Refactoring

- Used summaries to create patches and send them to developers
- Submitted patches to 5 applications
- Patches accepted in `libosip`, `patch` and `wget`

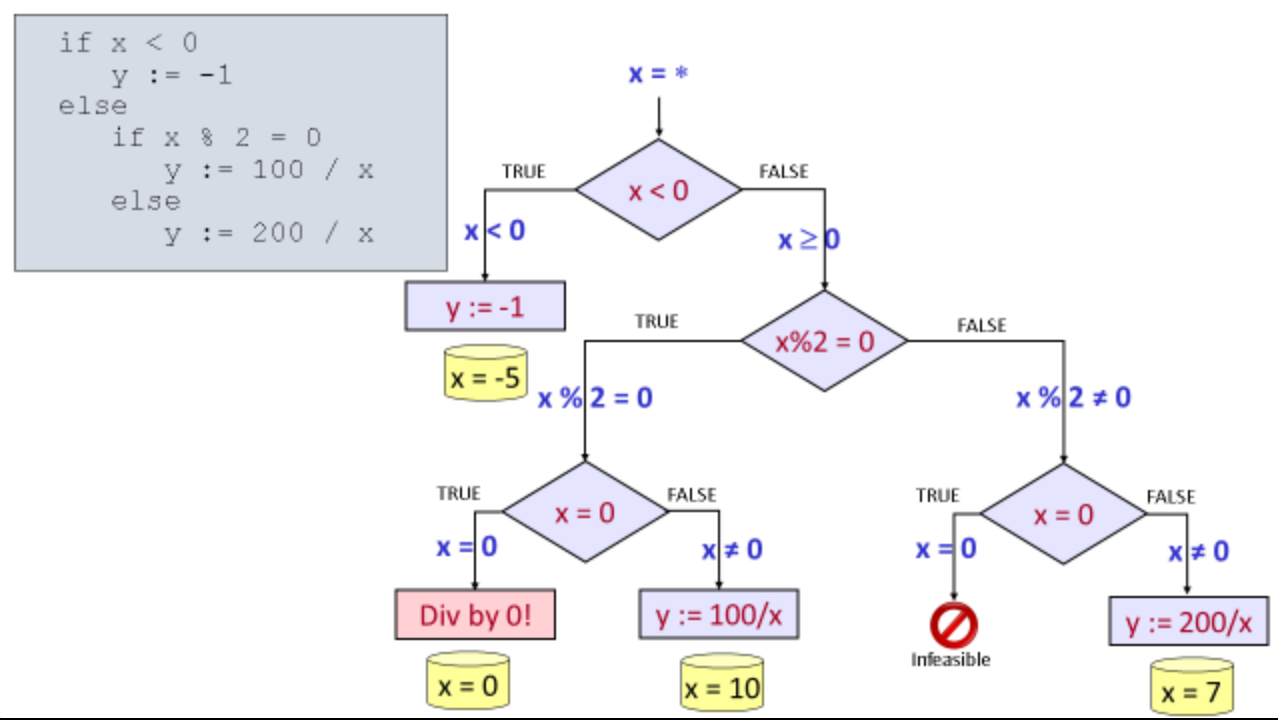
```
- for(; *tmp == ' ' || *tmp == '\t'; tmp++){
- }
- for(; *tmp == '\n' || *tmp == '\r'; tmp++){
- }                                  /* skip LWS */
+ tmp += strspn(tmp, " \t");
+ tmp += strspn(tmp, "\n\r");
```

Beyond Testing

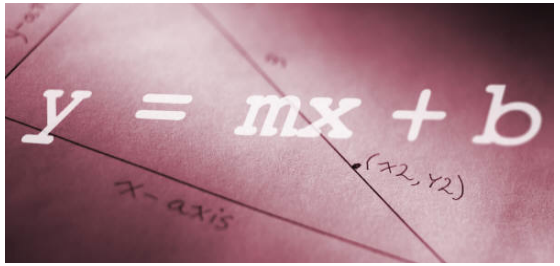
Symbolic execution has been shown useful in a variety of other areas:

- **Program debugging:** reproducing field failures
- **Program repair:** automatically generating patches
- **Refactoring:** generating equivalent code fragments
- **Equivalence checking:** reasoning about optimizations and refactorings
- **Automatic exploit generation:** construct actual security attacks
- **Verification:** network data planes, SIMD optimizations, image parsers, etc.
- **Side channel analysis:** quantifying information leakage
- **Reverse engineering:** constructing equivalent programs from binaries
- **Program synthesis:** generating programs that satisfy a specification
- **Document repair:** repairing electronic documents
- **Online games:** server-side verification of client behavior
- **+ many more**

```
if x < 0
   y := -1
else
   if x % 2 = 0
      y := 100 / x
   else
      y := 200 / x
```



Ongoing Challenges



Path Explosion

Constraint Solving

Fifteen Years Later

KLOVER

Miasm

PyExZ3

SymDroid

PathGrind

jCUTE

SAGE

Otter

SymJS

BinSE

CREST

Jalangi2

BINSEC

CUTE

Symbolic PathFinder

angr

DART

Kite

LDSE

Pex

Rubyx

JDart

CATG

CiVL

Mayhem

S²E