Advanced Software Testing and Debugging (CS598)
Symbolic Execution

Fall 2020

Lingming Zhang

Brief history

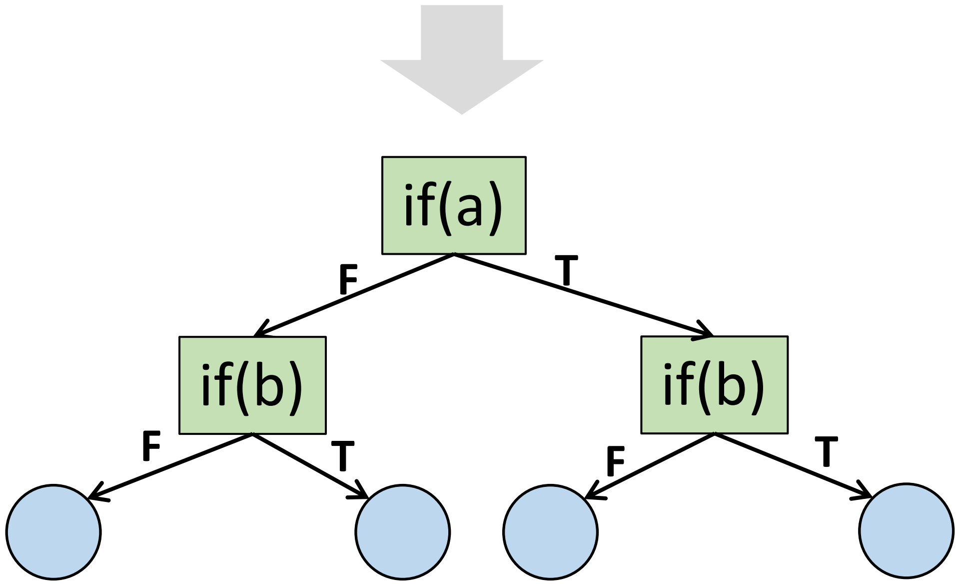
- **1976:** A system to generate test data and symbolically execute programs (Lori Clarke)
- **1976:** Symbolic execution and program testing (James King)

- **2005-present:** practical symbolic execution
  - Using SMT solvers
  - Heuristics to control exponential explosion
  - Heap modeling and reasoning about pointers
  - Environment modeling
  - Dealing with solver limitations

Program execution paths

- **Program** can be viewed as binary tree with possibly infinite depth
- Each **node** represents the execution of a conditional statement
- Each **edge** represents the execution of a sequence of non-conditional statements
- Each **path** in the tree represents an equivalence class of inputs

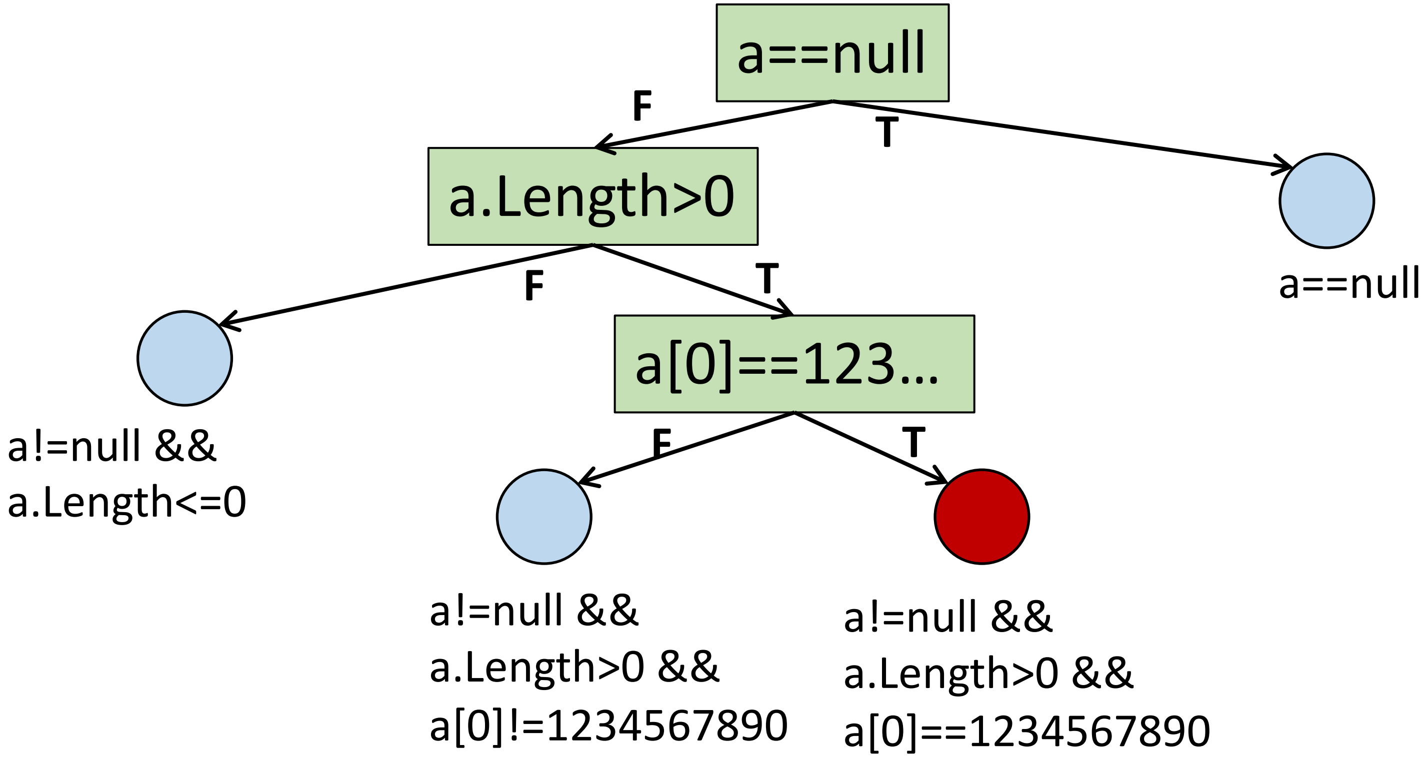
```
if (a)
    ...
if (b)
    ...
```



Example

Code under test

```
void CoverMe(int[] a) {
   if (a == null)
      return;
   if (a.Length > 0)
      if (a[0] == 1234567890)
         throw new Exception("bug");
}
```



Random testing?

Code under test

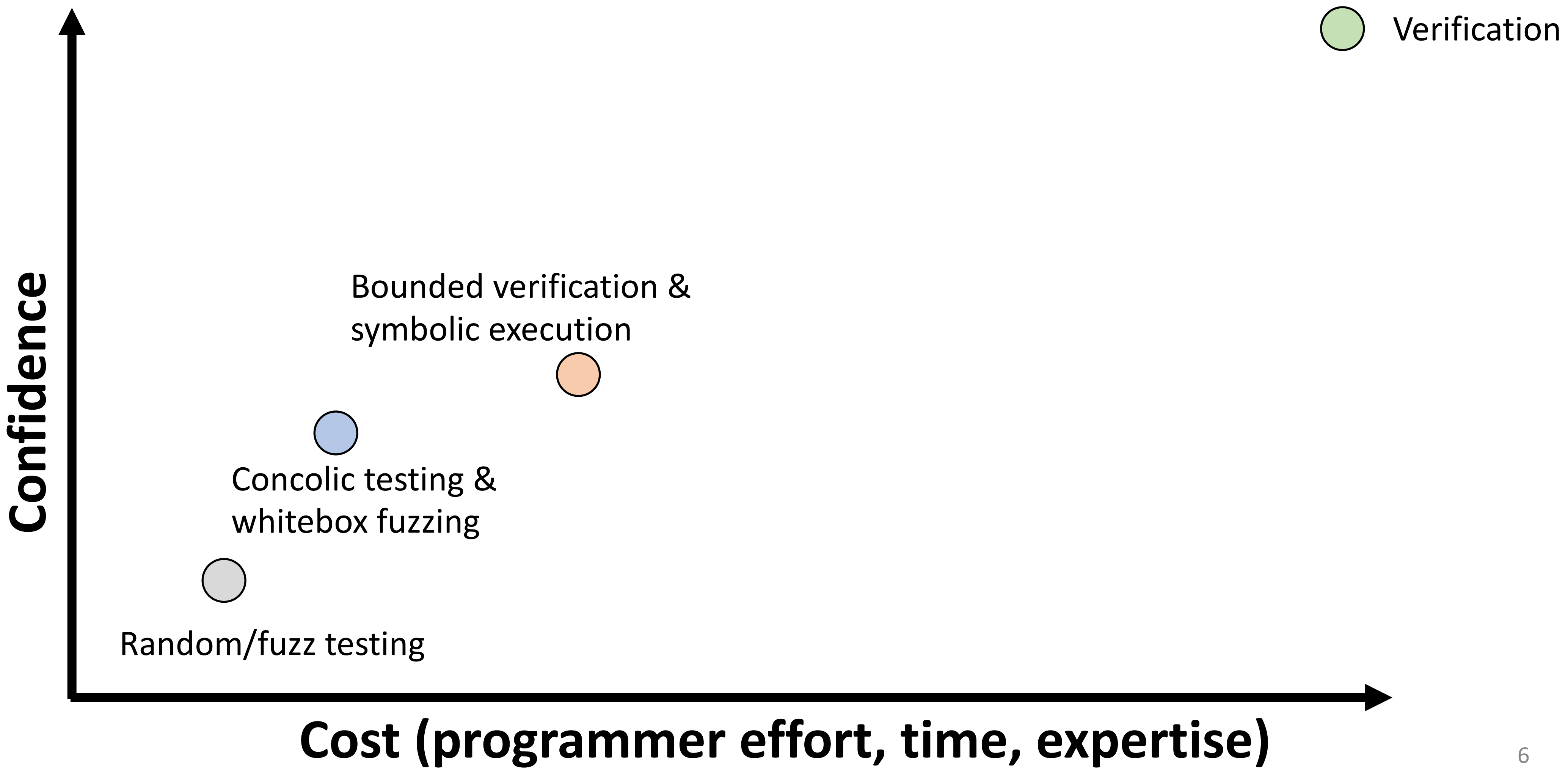
```
void CoverMe(int[] a) {
  if (a == null)
    return;
  if (a.Length > 0)
    if (a[0] == 1234567890)
      throw new Exception("bug");
}
```

- Random Testing
  - Generate random inputs
  - Execute the program on those (concrete) inputs
- Problem:
  - Probability of reaching error could be astronomically small

Probability of ERROR for the gray branch:

$1/2^{32} \approx 0.000000023\%$

The spectrum of program testing/verification



This class

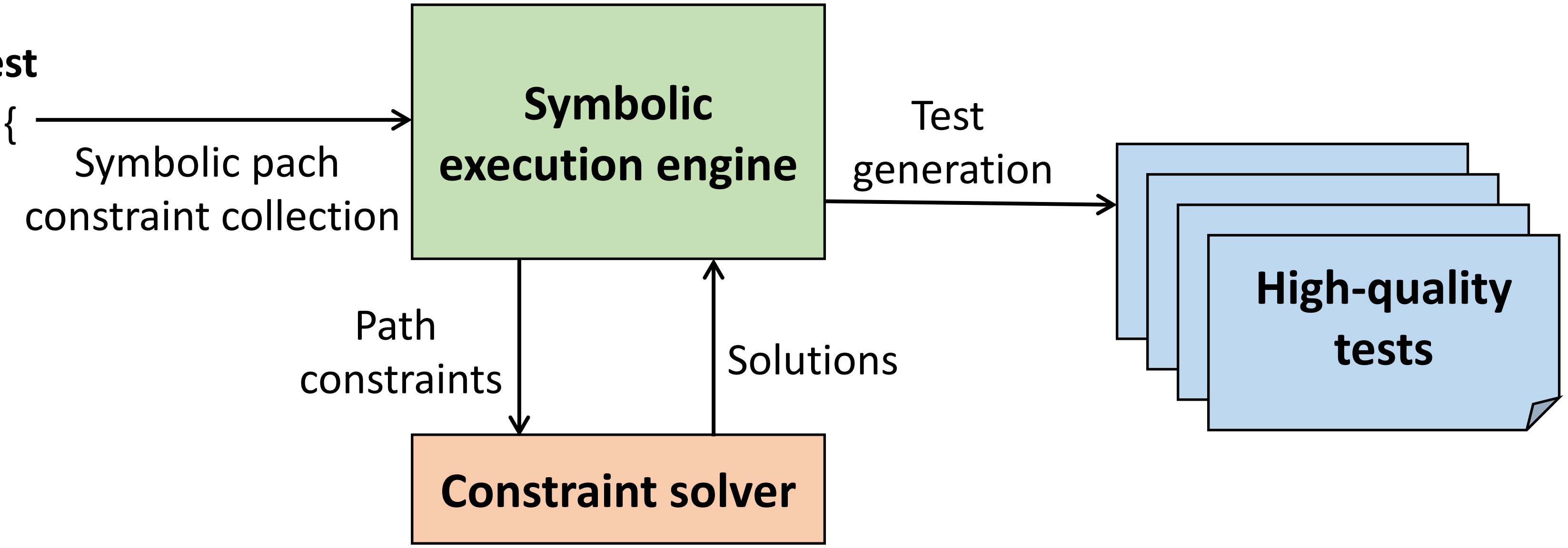
- KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (OSDI'08)
- Hybrid Concolic Testing (ICSE'07)

Symbolic execution

- Symbolic Execution
  - Use symbolic values for inputs
  - Execute program symbolically on symbolic input values
  - Collect symbolic path constraints (PCs)
  - Use SMT/SAT solvers to check if a branch can be taken

Code under test

```
int foo(int i) {
   int j=2*I;
   i=i++;
   i=i*j;
   if(i<1)
      i=-i;
   return i;
}
```



Symbolic execution: example

Code under test

```
int foo(int i) {
    int j=2*I;
    i=i++;
    i=i*j;
    if(i<1)
        i=-i;
    return i;
}
```

Concrete execution

i=1
i=1, j=2*1
i=1+1
i=2*2
4<1=false

return 4;

Symbolic execution

i=$\mathbf{I_0}$
PC=true, i= $\mathbf{I_0}$, j=2* $\mathbf{I_0}$
PC=true, i= $\mathbf{I_0}$ +1, j=2* $\mathbf{I_0}$
PC=true, i=($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$, j=2* $\mathbf{I_0}$
PC=true, i=($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$, j=2* $\mathbf{I_0}$

T

PC=($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$<1, i=-($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$, …
PC=($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$<1, **return** -($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$

Generated test1:
i=$\mathbf{I_0}$=0

F

PC=($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$>=1, **return** ($\mathbf{I_0}$ +1)*2$\mathbf{I_0}$

Generated test2:
i=$\mathbf{I_0}$=1

Symbolic execution: bug finding

- How to extend symbolic execution to catch non-crash bugs?
- Add dedicated checkers at dangerous code locations!
  - Divide by zero example: **y = x / z** where **x** and **z** are symbolic variables and assume current PC is **p**
    - Check if **z==0&&p** is possible!

```
int foo(int i) {
   int j=2*I;
   i=i++;
   i=i*j;
   if(i<1)
      i=-i;
   i=j/i;
   return i;
}
```

Code under test

True branch → **PC**=(I_0 +1)*2I_0<1, i=-(I_0 +1)*2I_0, i=0 → (I_0 +1)*2I_0<1 ∧ -(I_0 +1)*2I_0=0 → i= I_0 = 0 or i=I_0 = -1 **Trigger the bug!**

False branch → **PC**=(I_0 +1)*2I_0>=1, i= (I_0 +1)*2I_0, i=0 → (I_0 +1)*2I_0>=1 ∧ (I_0 +1)*2I_0=0 → **UNSAT** **Always safe!**

We can easily generate a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, …)

Challenges: path explosion

- Interleaving two search heuristics:
  - **Random Path Selection**: when a branch point is reached, the set of states in each subtree has equal probability of being selected
  - **Coverage-Optimized Search**: selects states likely to cover new code in the immediate future, based on
    - The minimum distance to an uncovered instruction
    - The call stack of the state
    - Whether the state recently covered new code

Challenges: optimizing SMT queries

- Expression rewriting
  - Simple arithmetic simplifications (x * 0 = 0)
  - Strength reduction (x * 2n = x << n)
  - Linear simplification (2 * x - x = x)
- Constraint set simplification
  - x < 10 && x = 5 --> x = 5
- Implied value concretization
  - x + 1 = 10 --> x = 9
- Constraint independence
  - i<j && j < 20 && k > 0 && i = 20 --> i<j && i<20 && i=20

Challenges: optimizing SMT queries (cont.)

- Counter-example cache
  - i < 10 && i = 10 (no solution)
  - i < 10 && j = 8 (satisfiable, with variable assignments i → 5, j → 8)
- Superset of unsatisfiable constraints
  - {i < 10, i = 10, j = 12} (unsatisfiable)
- Subset of satisfiable constraints
  - {i < 10} (satisfiable with i → 5, j → 8)
- Superset of satisfiable constraints
  - Same variable assignments might work

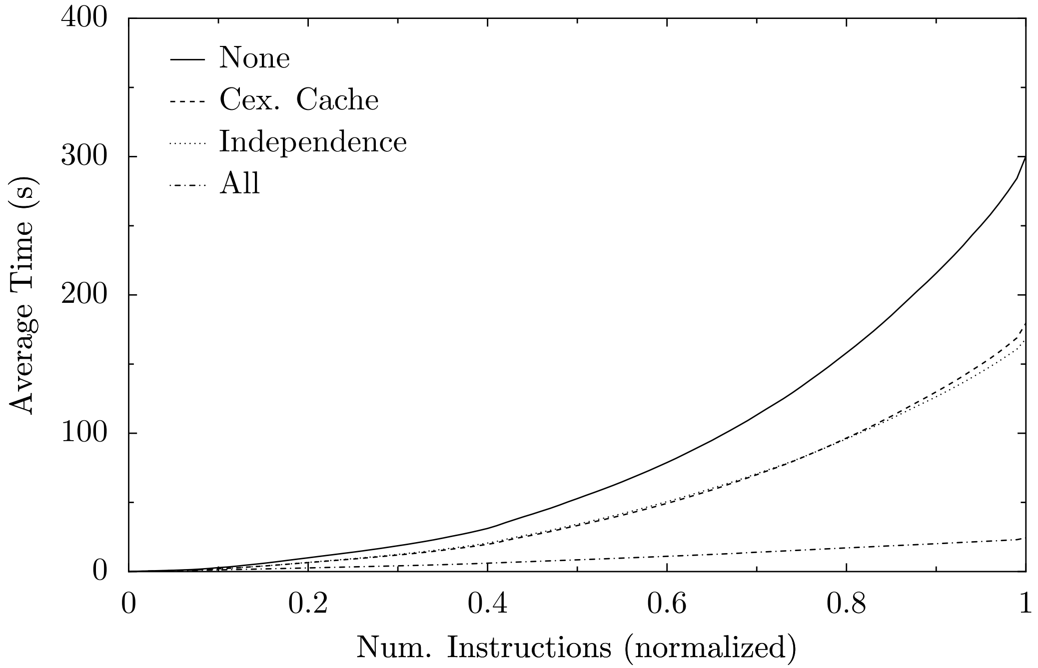


Figure 2: The effect of KLEE's solver optimizations over time, showing they become more effective over time, as the caches fill and queries become more complicated. The number of executed instructions is normalized so that data can be aggregated across all applications.

Challenges: environment modeling

```
int fd = open("t.txt", O_RDONLY);
```

- If all arguments are concrete, forward to OS directly

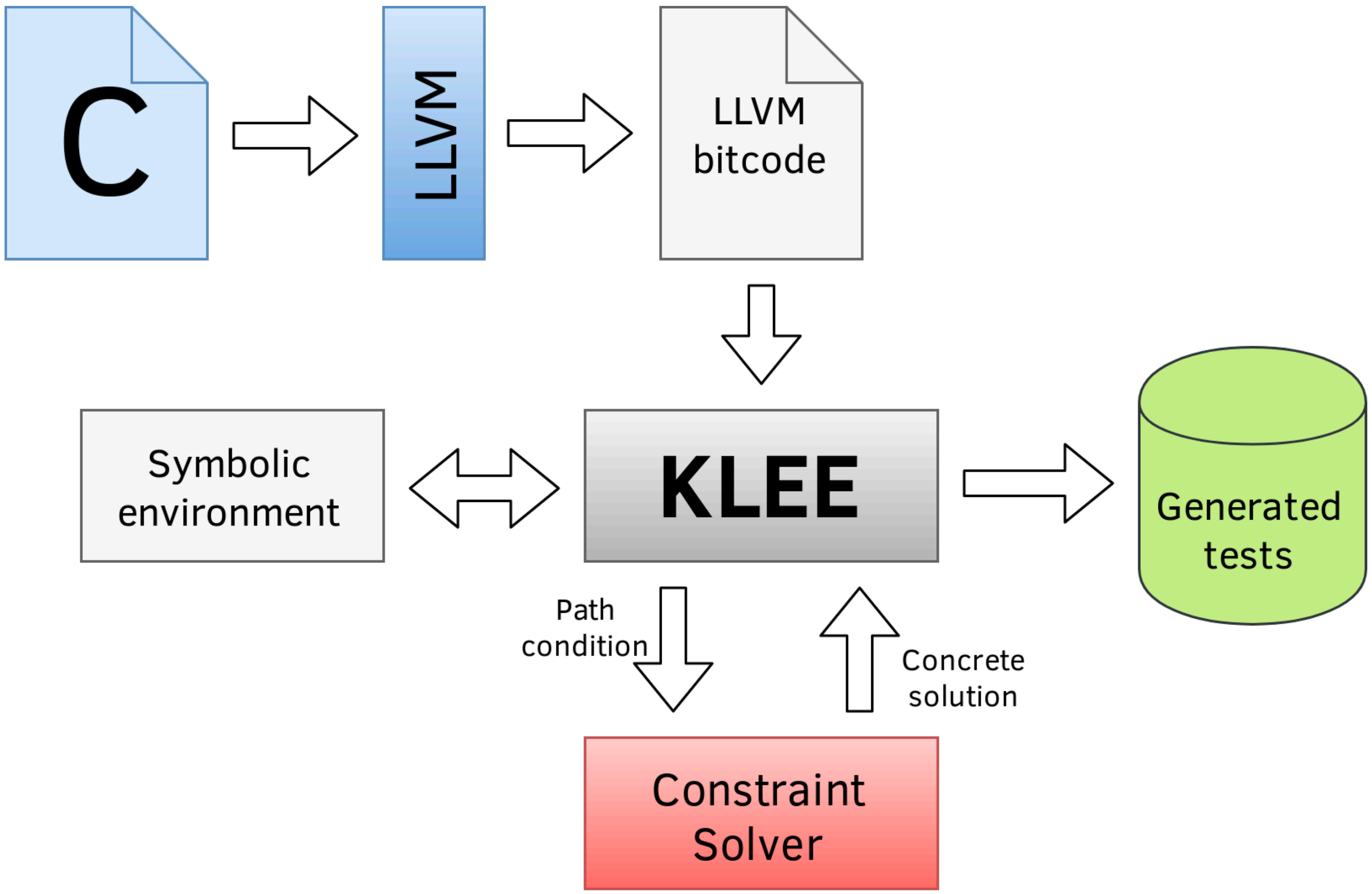
```
int fd = open(sym_str, O_RDONLY);
```

- Otherwise, provide models that can handle symbolic files
  - Goal is to explore all possible interactions with the environment
- About 2,500 LoC to define simple models for roughly 40 system calls
  - e.g., open, read, write, stat, lseek, ftruncate, ioctl

```
ssize t read(int fd, void *buf, size_t count) {
   …
   struct klee_fd *f = &fds[fd];
   …
   /* sym files are fixed size: don't read
beyond the end. */
   if (f->off >= f->size)
      return 0;
   count = min(count, f->size − f->off);
   memcpy(buf, f->file_data + f->off, count);
   f->off += count;
   return count;
}
```

Sketch of KLEE's model for read()

KLEE implementation



Benchmarks

- 89 programs in GNU **Coreutils** (version 6.10), roughly 80,000 lines of library code and 61,000 lines in the actual utilities, including ones
  - Managing the file system (e.g., ls, dd, chmod)
  - Displaying and configuring system properties (e.g., logname, printenv)
  - Controlling command invocation (e.g., nohup, nice, env)
  - Processing text files (e.g., sort, od, patch)
- Two other UNIX utility suites: **Busybox**, a widely-used distribution for embedded systems, and the latest release for **Minix**
- The **HiStar** operating system kernel

Coverage

	COREUTILS		BUSYBOX	
Coverage (w/o lib)	KLEE tests	**Devel. tests**	KLEE tests	**Devel. tests**
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

Coreutils bugs detected

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

Busybox bugs detected

```
date -I
ls --co
chown a.a -
kill -l a
setuidgid a ""
printf "% *" B
od t1.txt
od t2.txt
printf %
printf %Lo
tr [
tr [=
tr [a-z
```

t1.txt: a
t2.txt: A
t3.txt: \t\n

```
cut -f t3.txt
install --m
nmeter -
envdir
setuidgid
envuidgid
envdir -
arp -Ainet
tar tf_ /
top d
setarch "" ""
<full-path>/linux32
<full-path>/linux64
hexdump -e ""
ping6 -
```

Figure 10: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in BUSYBOX. When multiple applications crash because of the same shared (buggy) piece of code, we group them by shading.

Inconsistencies between Coreutils and Busybox

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[infinite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/: Is a directory"`
`split /`	`"/: Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[ 0 ‘‘<’’ 1 ]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

Table 3: Very small subset of the mismatches KLEE found between the BUSYBOX and COREUTILS versions of equivalent utilities. The first three are serious correctness errors; most of the others are revealing missing functionality.

Inconsistencies between Coreutils and Busybox: how?

```
unsigned mod_opt(unsigned x, unsigned y){
   if((y&-y)==y) // power of two?
      return x& (y-1);
   else
      return x%y;
}
```

Implementation 1

```
unsigned mod_opt(unsigned x, unsigned y){
   return x%y;
}
```

Implementation 2

```
int main(){
   unsigned x, y;
   make_symbolic(&x, sizeof(x));
   make_symbolic(&y, sizeof(y));
   assert(mod(x,y) == mod_opt(x,y));
   return 0;
}
```

Every assertion can be treated as a branch statement with two outgoing branches (i.e., hold or not); symbolic execution will try to cover both!

Discussion

- Strengths
- Limitations
- Future work

This class

- KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (OSDI'08)
- Hybrid Concolic Testing (ICSE'07)

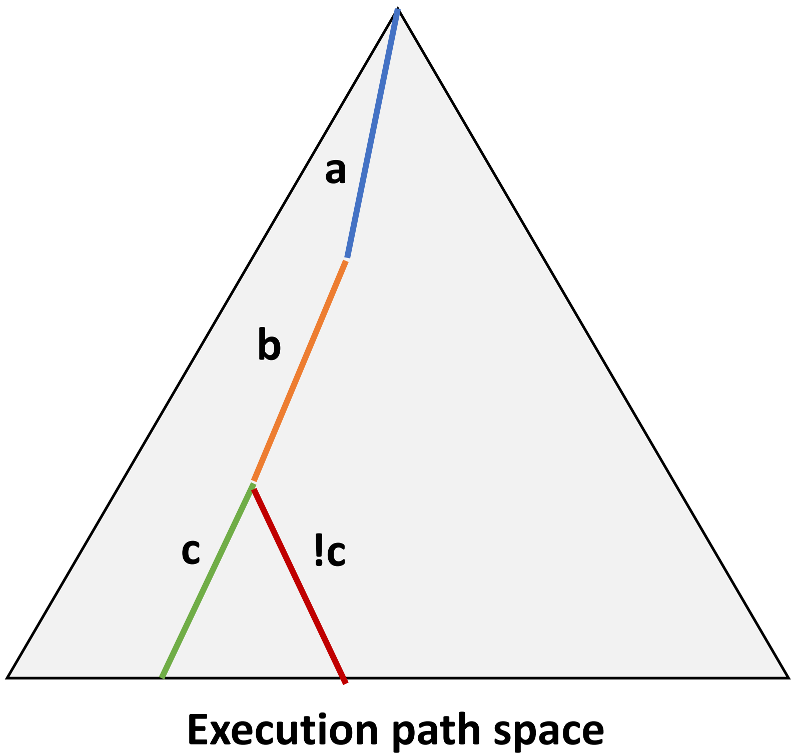
Symbolic execution: coverage problem

Symbolic execution may not reach deep into the execution tree. Specially when encountering loops

Solution: concolic execution

Concolic=Concrete+Symbolic

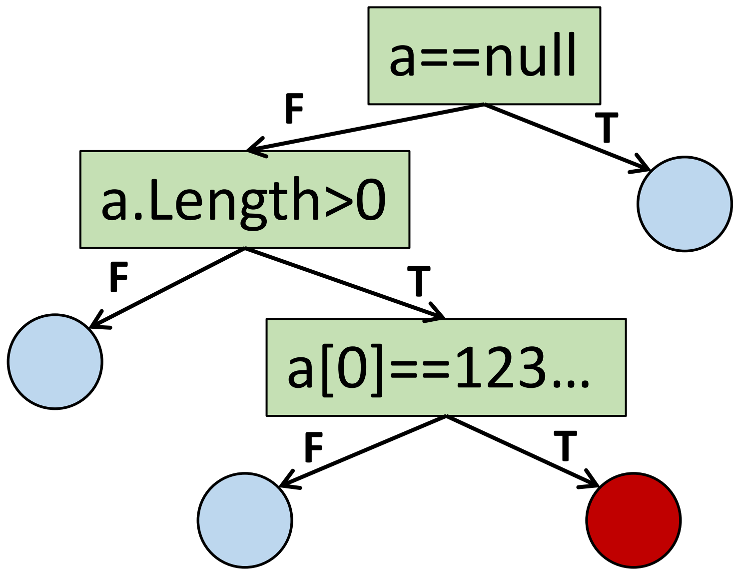
- Generate a random seed input to dive into the program execution tree
- Concretely execute the program with the random seed input and collect the path constraint, e.g., **a && b && c**
- In the next iteration, negate the last conjunct to obtain the constraint: **a && b && !c**
- Solve it to get input to the path which matches all the branch decisions except the last one



Concolic execution

Code under test

```
void CoverMe(int[] a) {
  if (a == null)
    return;
  if (a.Length > 0)
    if (a[0] == 1234567890)
      throw new Exception("bug");
}
```



Executed paths

Choose next patch

Solve **Monitor**

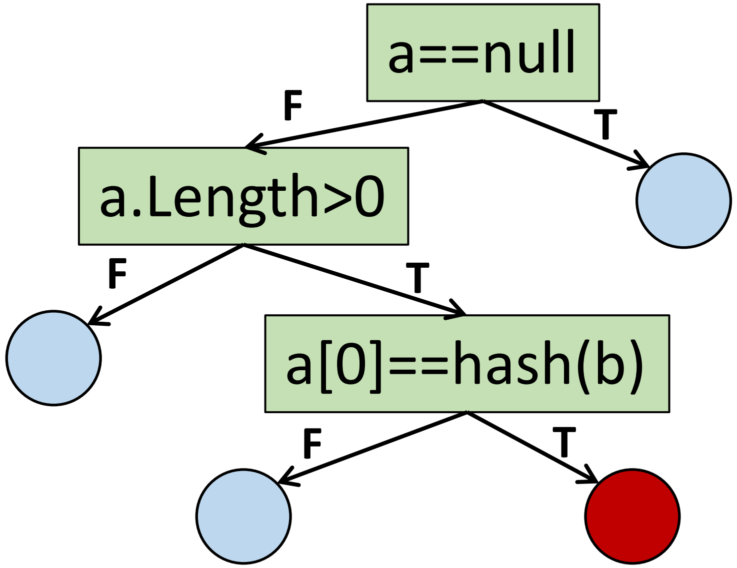
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{1}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890	{123…}	a!=null && a.Length>0 && a[0]==1234567890

DONE, no path left!

Concolic execution: another example

Code under test

```
void CoverMe(int[] a, int b) {
  if (a == null)
    return;
  if (a.Length > 0)
    if (a[0] == hash(b))
      throw new Exception("bug");
}
```



Executed paths

Choose next patch

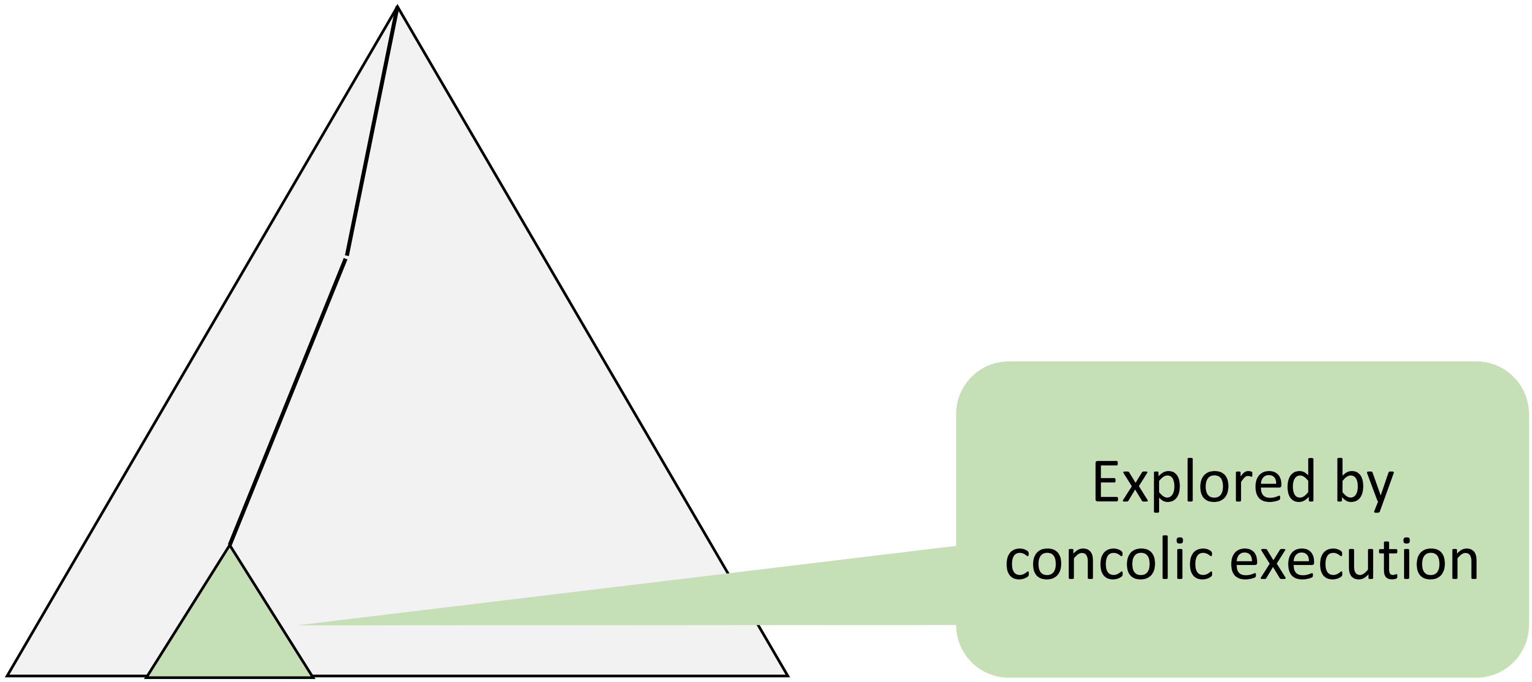
Solve **Monitor**

Constraints to solve	**Data**	**Observed constraints**
	null, 0	a==null
a!=null	{}, 0	a!=null && !(a.Length>0)
a!=null && a.Length>0	{1}, 0	a!=null && a.Length>0 && a[0]!=hash(b)
a!=null && a.Length>0 && a[0]==hash(0)	{434…}, 0	a!=null && a.Length>0 && a[0]==434…

Concretized!

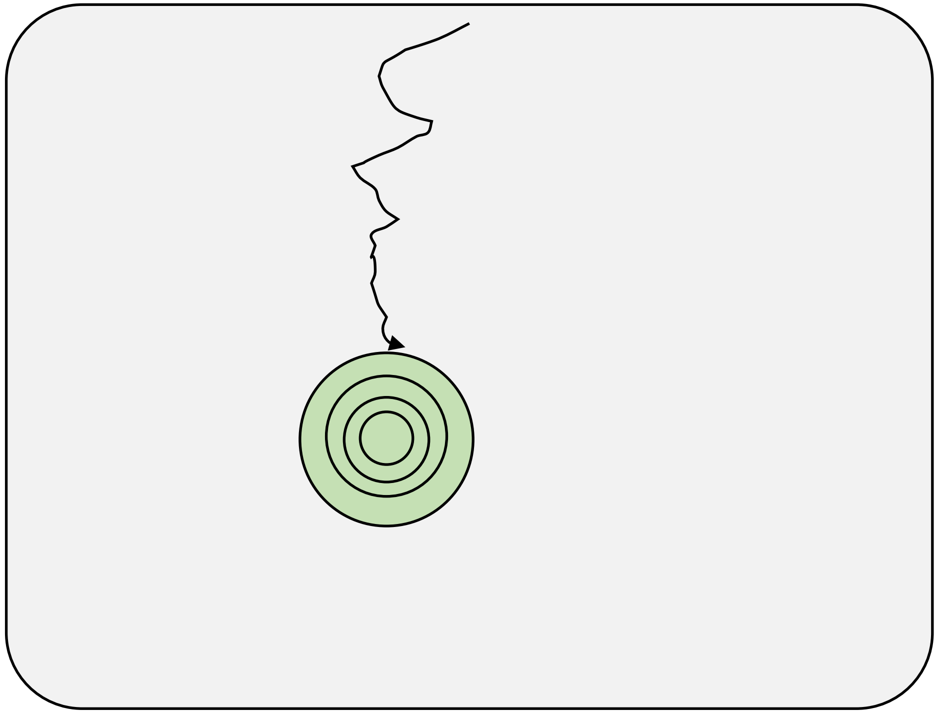
DONE, no path left!

Limitations

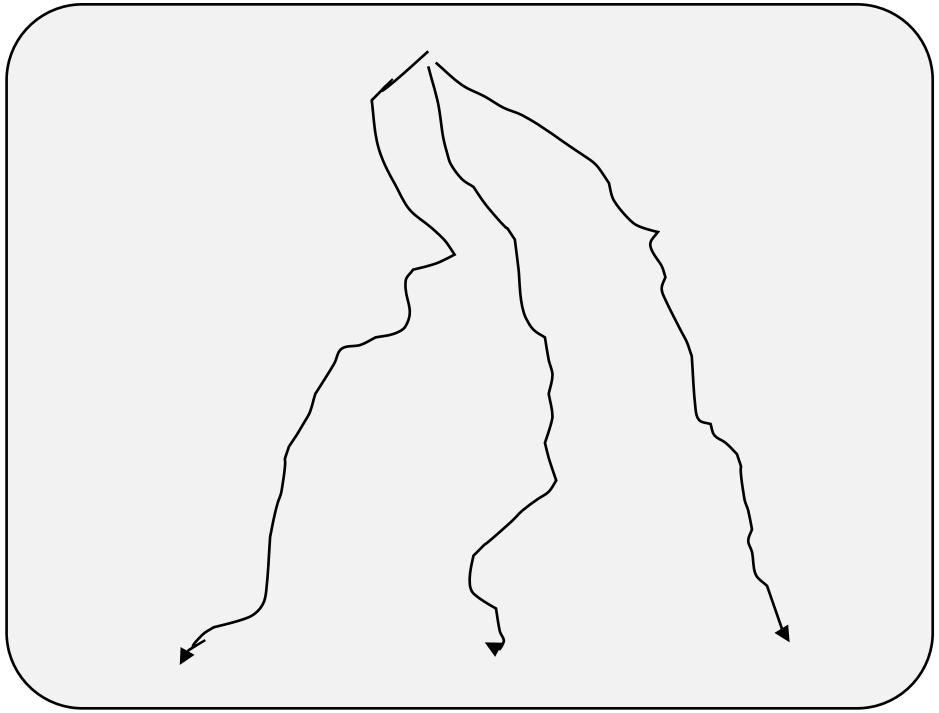


Execution path space

Limitations: a comparative view



Concolic testing: wide and shallow



Random testing: narrow and deep

Limitations: example

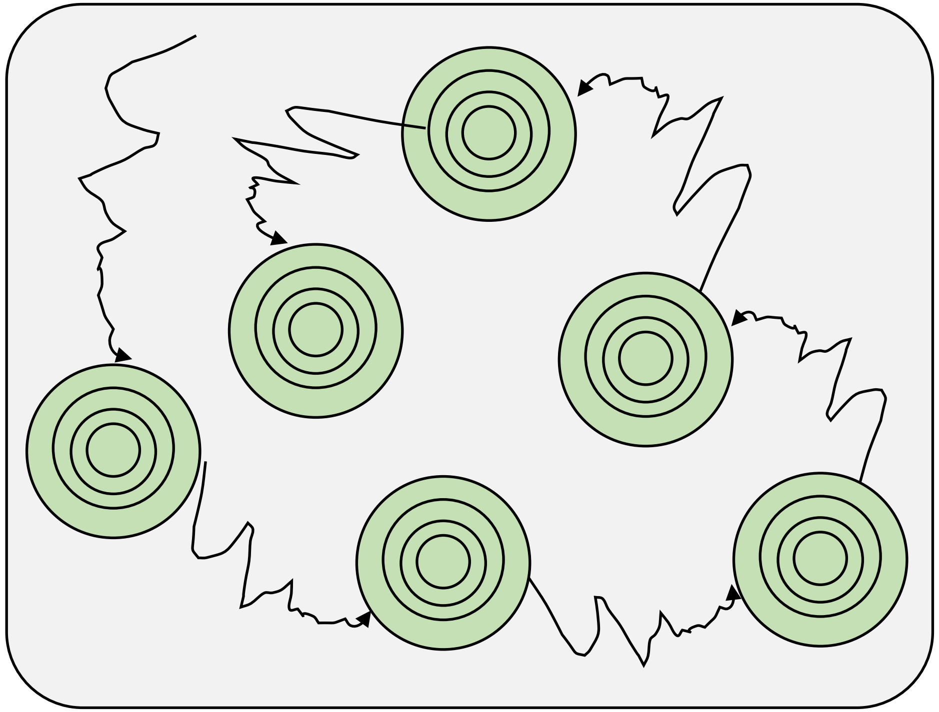
```
Example ( ) {
  state = 0;
  while(1) {
    s = input();
    c = input();
    if(c==':' && state==0)
      state=1;
    else if(c=='\n' && state==1)
      state=2;
    else if (s[0]=='U' &&
      s[1]=='I' &&
      s[2]=='U' &&
      s[3]=='C' &&
      state==2) {
        COVER_ME:;
    }
  }
}
```

- **COVER_ME** can be hit on an input sequence
  - **s = 'UIUC'**
  - **c : ':' '\n'**
- **Random testing** can get to **state = 2**, but difficult to get **'UIUC'** as a sequence
  - Probability: $\mathbf{1/(2^8)^4 \approx 2.3X10^{-10}}$
- **Concolic testing** can generate **'UIUC'**, but explores many paths to **state = 2**

Similar code structure in

- Text editors (vi)
- Parsers (lexer)
- Event-driven programs (GUI)

Hybrid concolic testing



while (not required coverage) {
    while (not saturation)
        perform random testing;
    Checkpoint;
    while (not increase in coverage)
        perform concolic testing;
    Restore;
}

Interleave **random testing** and **concolic testing**
for **deep&broad search** to increase coverage

Hybrid concolic testing: example

```
Example ( ) {
  state = 0;
  while(1) {
    s = input();
    c = input();
    if(c==':' && state==0)
      state=1;
    else if(c=='\n' && state==1)
      state=2;
    else if (s[0]=='U' &&
      s[1]=='I' &&
      s[2]=='U' &&
      s[3]=='C' &&
      state==2) {
        COVER_ME:;
    }
  }
}
```

- **Random phase:** '$', '&', '-', '6', ':', '%', '^', '\n', 'x', '~' …
  - Saturates after many (~10000) iterations
  - In less than 1 second
  - **COVER_ME** is not reached

- **Concolic phase:** s[0]='U', s[1]='I', s[2]='U', s[3]='C'
  - Reaches **COVER_ME**!

Implementation

- An extension on the CUTE:
  - A concolic execution engine for C
  - Code instrumentation via **CIL**, a framework for parsing and transforming C programs[1]
  - Constraint solving via **lp_solve**, a library for integer linear programming[2]

1 https://github.com/cil-project/cil
2 http://lpsolve.sourceforge.net/5.5/

Testing red-black tree

	Branch Coverage in Percentage		
Seed	Random Testing	Concolic Testing	Hybrid Concolic Testing
523	32.27	52.48	66.67
7487	32.27	52.48	67.02
6726	32.27	52.48	66.67
5439	32.27	52.48	67.73
4494	32.27	52.48	69.86
Average	32.27	52.48	67.59

Table 1. Results of Testing Red-Black Tree

Testing Vim editor

	Branch Coverage in Percentage		
Seed	Random Testing	Concolic Testing	Hybrid Concolic Testing
877443	8.01	21.43	41.93
67532	8.16	21.43	40.39
98732	8.72	21.43	33.67
32761	7.80	21.43	35.45
28683	9.75	21.43	40.53
Average	8.17	21.43	37.86

Table 2. Results of Testing the VIM Test Editor

Discussion

- Strengths
- Limitations
- Future work

Symbolic execution engines you may want to try

- C family: KLEE (http://llvm.org/)
- C#: Pex/IntelliTest (http://research.microsoft.com/en-us/projects/pex/)
- Java: Java PathFinder (https://github.com/SymbolicPathFinder/jpf-symbc)
- JavaScript: Jalangi2 (https://github.com/Samsung/jalangi2)
- Binaries (x86, ARM, ...): S2E (https://s2e.systems/)

Further readings

- Koushik Sen, Darko Marinov, Gul Agha. CUTE: A Concolic Unit Testing Engine for C. 2005, FSE.
- Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler. EXE: Automatically Generating Inputs of Death. 2006, CCS.
- Patrice Godefroid, Michael Y. Levin, David Molnar. Automated Whitebox Fuzz Testing. 2008, NDSS.

Thanks and stay safe!