

An Introduction to KLEE

Cristian Cadar

Department of Computing
Imperial College London

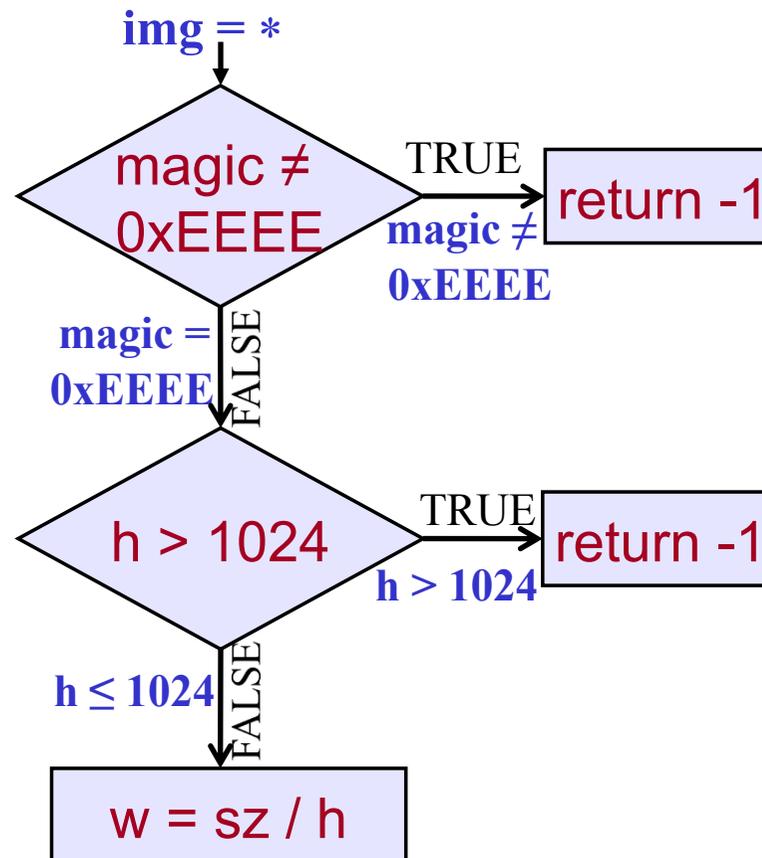


SOFTWARE RELIABILITY
GROUP

Toy Example

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

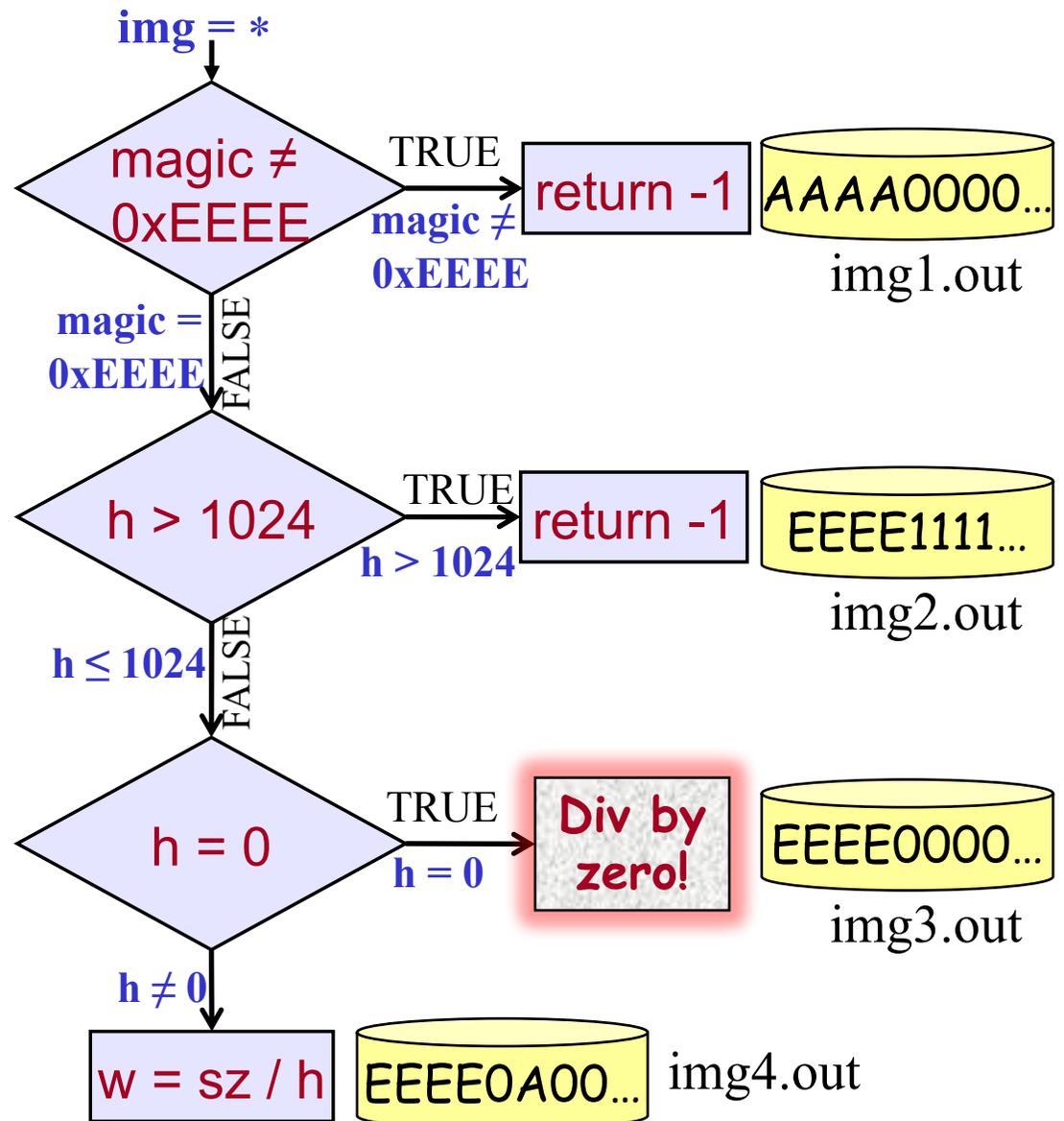
```
int main(int argc, char** argv) {  
    ...  
    image_t img = read_img(file);  
    if (img.magic != 0xEEEE)  
        return -1;  
    if (img.h > 1024)  
        return -1;  
    w = img.sz / img.h;  
    ...  
}
```



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;  
    ...  
}
```



All-Value Checks

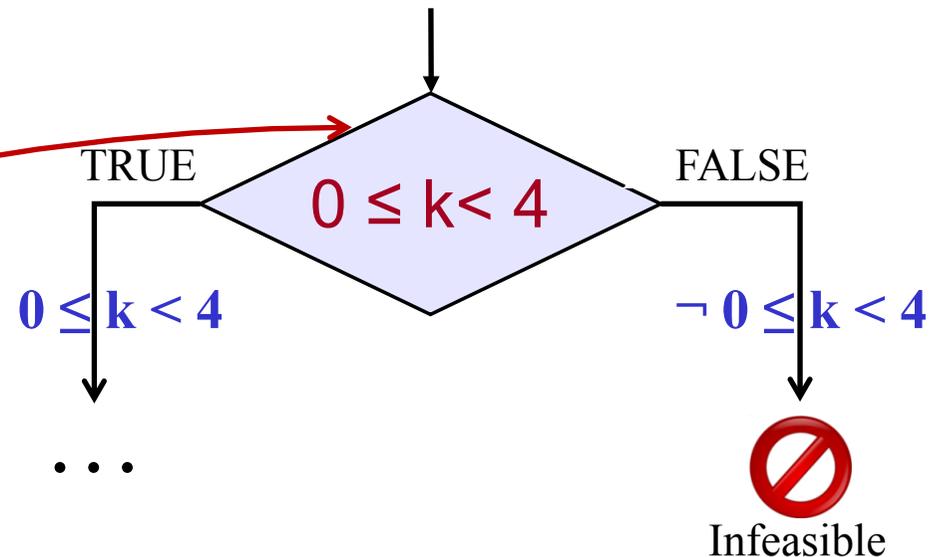
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {  
  int a[4] = {3, 1, 0, 4};  
  k = k % 4;  
  return a[a[k]];  
}
```



All-Value Checks

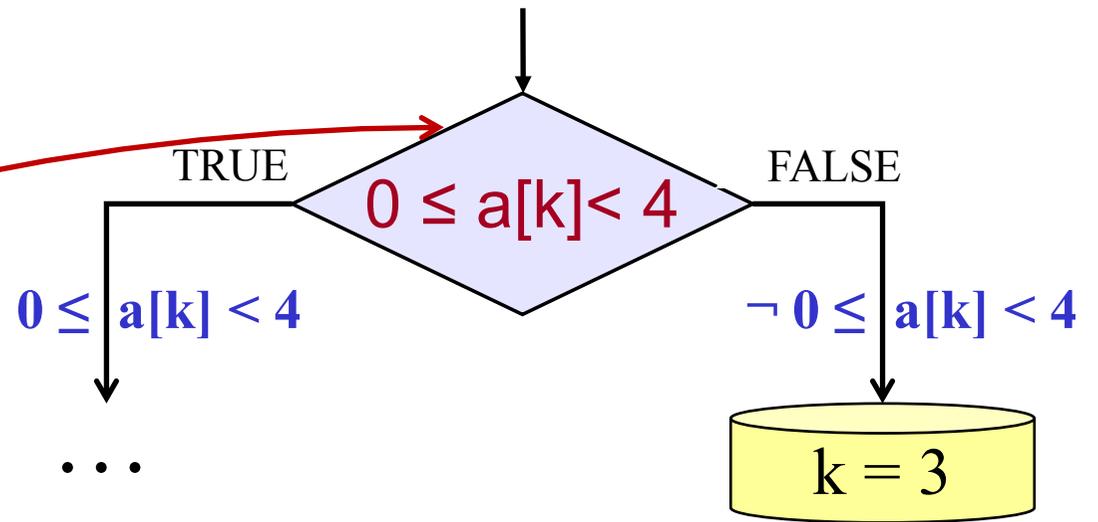
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {  
    int a[4] = {3, 1, 0, 4};  
    k = k % 4;  
    return a[a[k]];  
}
```



Buffer overflow!

KLEE

- Symbolic execution tool started as a successor to EXE
- Based on the LLVM compiler, primarily targeting C code
- Open-sourced in June 2009, now available on GitHub
- Active user base with over 300 subscribers on the mailing list and over 35 contributors listed on GitHub

Webpage: klee.github.io

Code: github.com/klee

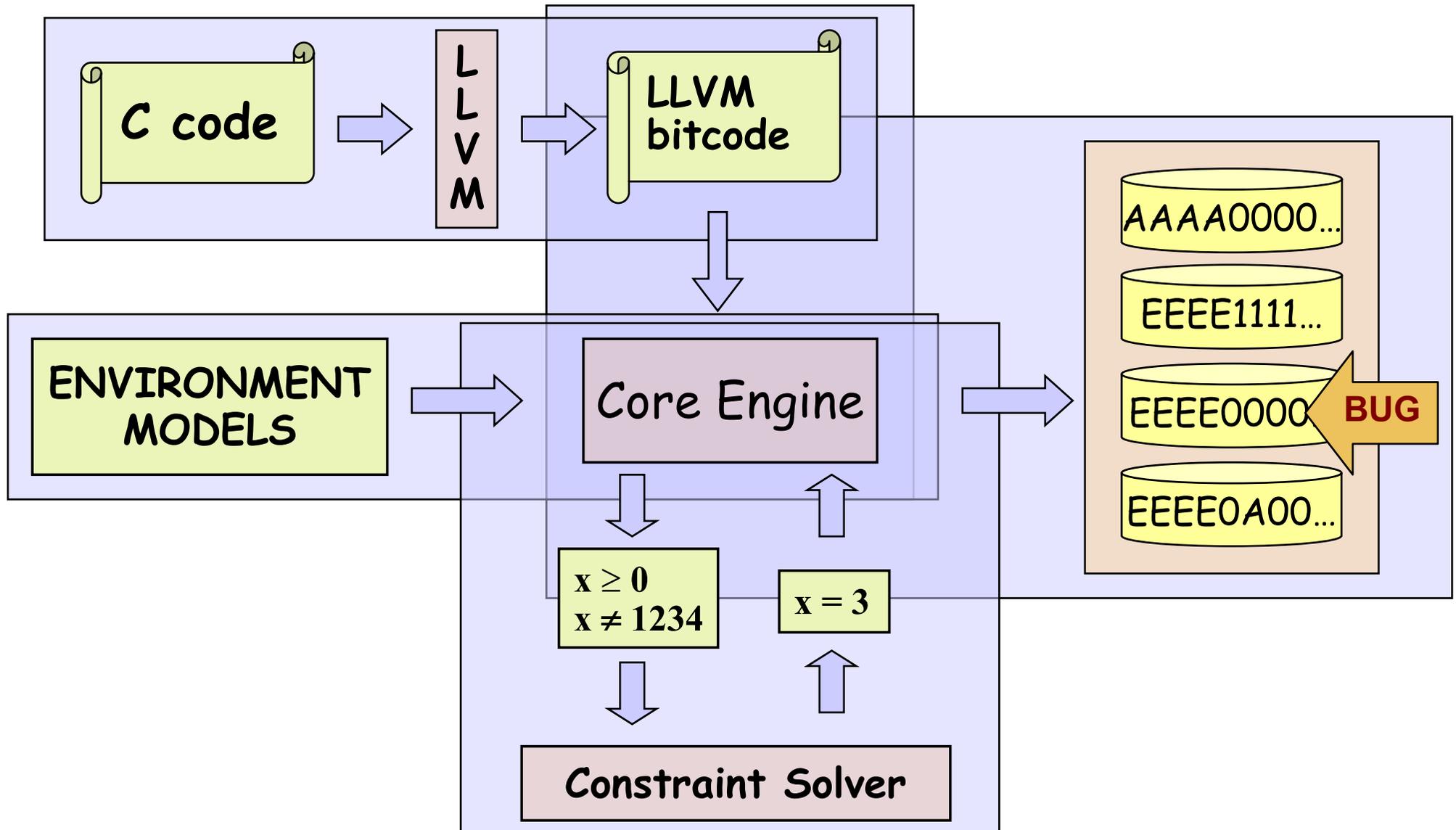
KLEE

- Extensible platform, used and extended by many groups in academia and industry, in the areas such as:
 - bug finding
 - high-coverage test input generation
 - exploit generation
 - automated debugging
 - wireless sensor networks/distributed systems
 - schedule memoization in multithreaded code
 - client-behavior verification in online gaming
 - GPU testing and verification, etc.

An incomplete list of publications and extensions available at:

klee.github.io/Publications.html

KLEE Architecture



KLEE Demo: Toy Image Viewer

```
// #include directives
struct image_t {
    unsigned short magic;
    unsigned short h, sz; // height, size
    char pixels[1018];
};
int main(int argc, char** argv) {
    struct image_t img;
    int fd = open(argv[1], O_RDONLY);
    read(fd, &img, 1024);

    if (img.magic != 0xEEEE)
        return -1;
    if (img.h > 1024)
        return -1;
    unsigned short w = img.sz / img.h;

    return w;
}
```

```
$ clang -emit-llvm -c -g image_viewer.c
$ klee --posix-runtime -write-pcs
    image_viewer.bc --sym-files 1 1024 A
...
KLEE: output directory = klee-out-1
(klee-last)
...
KLEE: ERROR: ... divide by zero
...
KLEE: done: generated tests = 4
```

KLEE Demo: Toy Image Viewer

```
$ cat klee-last/test000003.pc
...
array A-data[1024] : w32 -> w8 = symbolic
(query [
    ...
    (Eq 61166
      (ReadLSB w16 0 A-data))
    (Eq 0
      (ReadLSB w16 2 A-data))
    ...
  ]
)
```


KLEE Demo: All-Values Checks

```
int foo(unsigned k) {
    int a[4] = {3, 1, 0, 4};
    k = k % 4;
    return a[a[k]];
}

int main() {
    int k;
    klee_make_symbolic(&k, sizeof(k), "k");
    return foo(k);
}
```

```
$ clang -emit-llvm -c -g all-values.c
$ klee all-values.bc
...
KLEE: ERROR: /home/klee/all-values/all-
values.c:4: memory error: out of bound
pointer
...
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Running KLEE inside a Docker container

Step 1: Install Docker for Linux/MacOS/Windows

Step 2: `docker pull klee/klee`

Step 3: `docker run --rm -ti --ulimit='stack=-1:-1' klee/klee`

<http://klee.github.io/docker/>

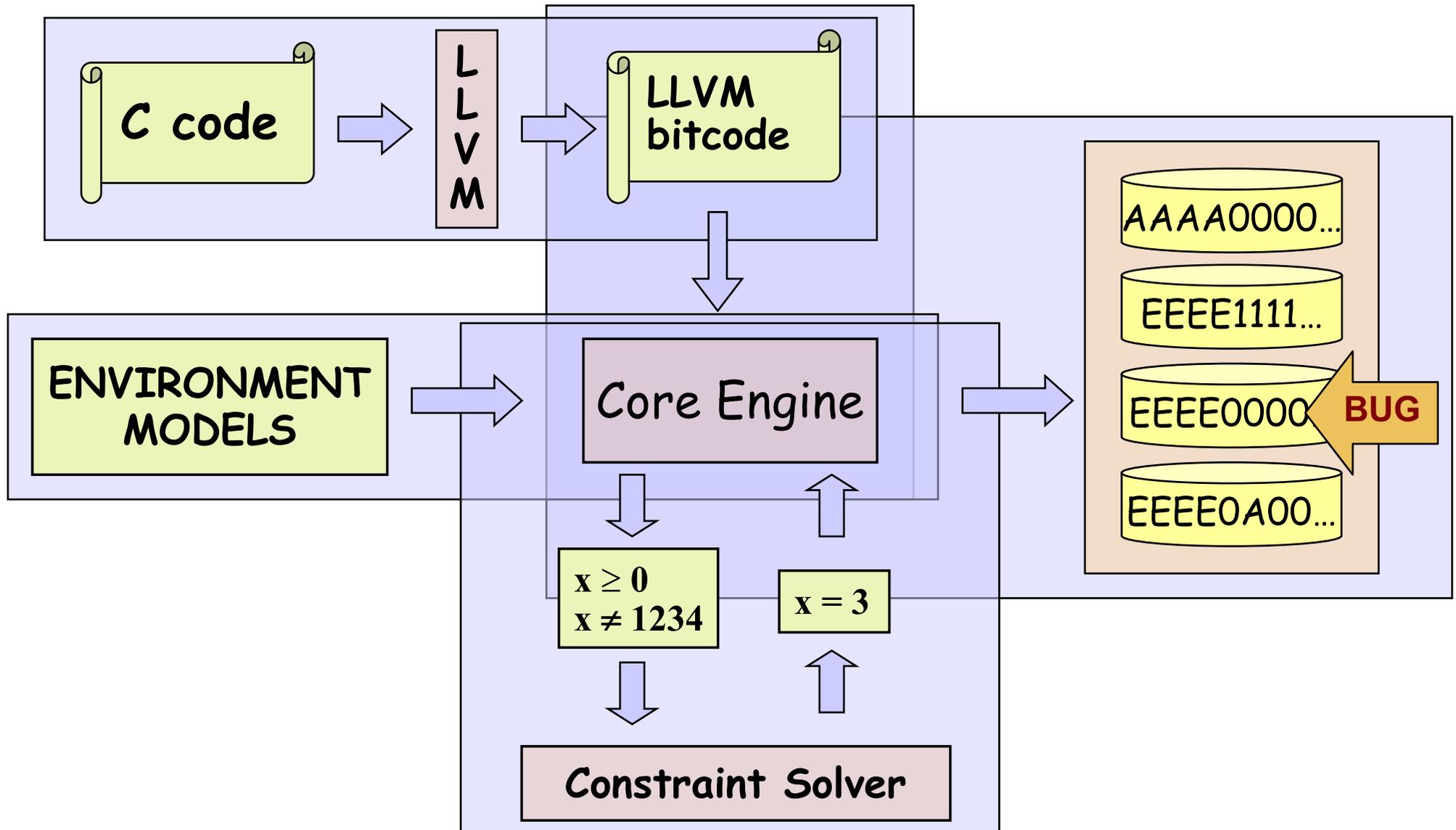
KLEE on the Web at <http://klee.doc.ic.ac.uk>

The screenshot displays the KLEE web interface. On the left, a sidebar contains a 'Tutorials' dropdown menu and a 'CURRENT FILE' section listing 'get_sign.c', 'get_sign.c', 'regexp.c', and 'maze.c'. The main area features the KLEE logo, a 'LOGIN' button, and a 'REGISTER' button. Below these are several configuration buttons: 'SYM. ARGS', 'SYM. FILES', 'SYM. INPUT', 'OPTIONS', and 'COVERAGE', each with a dropdown arrow. A prominent 'RUN KLEE' button is on the right. The central pane shows a C code editor with the following code:

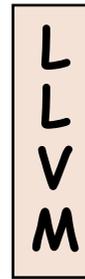
```
1  /*
2  * First KLEE tutorial: testing a small
3  * function
4  * http://klee.github.io/tutorials/testing-
5  * function/
6  */
7  int get_sign(int x) {
8      if (x == 0)
9          return 0;
10
11     if (x < 0)
12         return -1;
13     else
14         return 1;
15 }
16
17 int main() {
18     int a;
19     klee_make_symbolic(&a, sizeof(a), "a");
20     return get_sign(a);
21 }
```

On the right, the 'KLEE RESULTS' panel is visible, containing 'OUTPUT' and 'STATS' buttons, along with icons for email and cloud storage.

KLEE Architecture



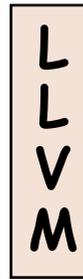
KLEE Architecture:



LLVM advantages:

- Mature framework, incorporated into commercial products by Apple, Google, Intel, etc.
- Elegant design patterns: analysis passes, visitors, etc.
- Single Static-Assignment (SSA) form with infinite registers (nice fit for symbolic execution)
- Lots of useful program analyses
- Well documented
- Several different front-ends, so KLEE could be extended to work with languages other than C

KLEE Architecture:



LLVM disadvantages

- Fast changing, not-backward compatible API!
 - KLEE is currently based on LLVM 3.4
 - But working on supporting newer versions
- Compiling to LLVM bitcode is still not trivial, but it's getting better:
 - `make CC="clang -emit-llvm"`
 - LLVM Gold Plugin <http://llvm.org/docs/GoldPlugin.html>
 - Whole-Program LLVM <https://github.com/travitch/whole-program-llvm>

KLEE Architecture:

L
L
V
M

KLEE runs LLVM, not C code!

```
#include <stdio.h>
int main() {
    int x;
    klee_make_symbolic(&x, sizeof(x), "x");

    if (x > 0)
        printf("x\n");
    else printf("x\n");

    return 0;
}
```

```
$ clang -emit-llvm -c -g code.c
$ klee code.bc
```

...

x

```
KLEE: done: total instructions = 6
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

KLEE Architecture:

Core Engine

The core engine implements symbolic execution exploration.

...	Interpreter	...
Memory	Core Engine	Stats
...	Searchers	...

KLEE Architecture:

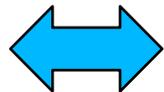
Interpreter

Core Engine

- Works as a mixed concrete/symbolic interpreter for LLVM bitcode

```
Instruction *i = ki->inst;
switch (i->getOpcode()) {
  case Instruction::Ret:
    ...
  case Instruction::Br:
    // if both sides feasible, fork
    ...
}
```

\$./program



\$ klee program.bc

Paths and Execution States

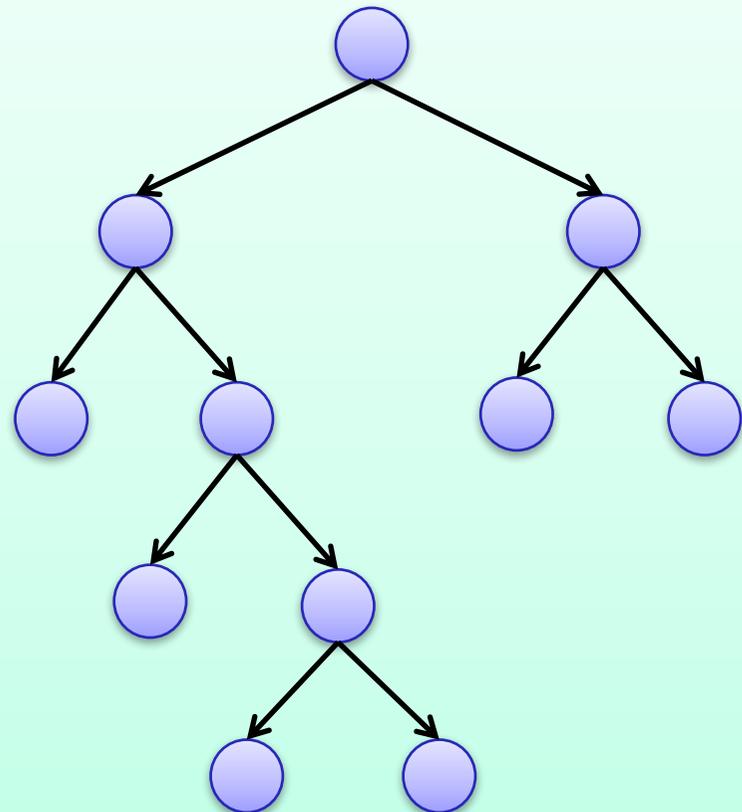
- Each path represented by an *ExecutionState*, with KLEE acting as an OS for ExecutionStates

ExecutionState

- PC
- Stack
- Address space
- List of sym objects
- Path constraints
- etc.

- Fork implemented by object-level COW

Tree of ESs



Search Heuristics in KLEE

Core Engine

Searchers

- Basic search heuristics such as BFS and DFS

```
klee --search=dfs program.bc
```

- Coverage-optimized search (**--search=nurs:md2u**)
 - Select path closest to an uncovered instruction
- Random-state search (**--search=random-state**)
 - Randomly select a pending state/path
- Random-path search (**--search=random-path**)
 - Described in the lecture
- etc.

Combining Search Heuristics

Core Engine

Searchers

KLEE can also use multiple heuristics in a round-robin fashion, to protect against individual heuristics getting stuck in a local maximum.

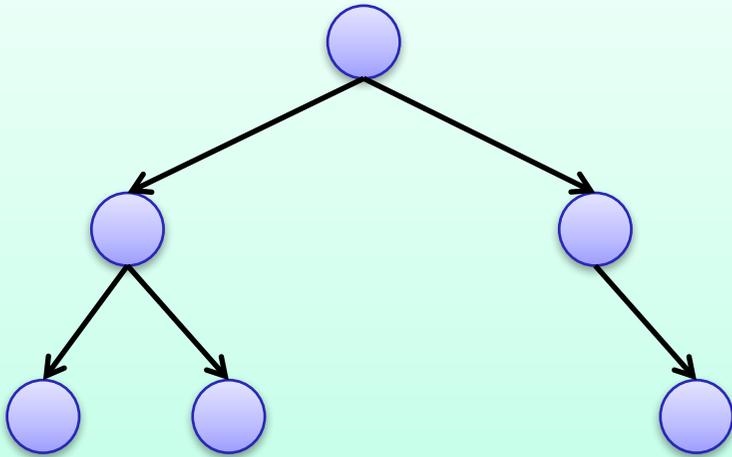
```
klee --search=nurs:md2u --search=dfs  
--search=random-path ...
```

New Search Heuristics

Core Engine

Searchers

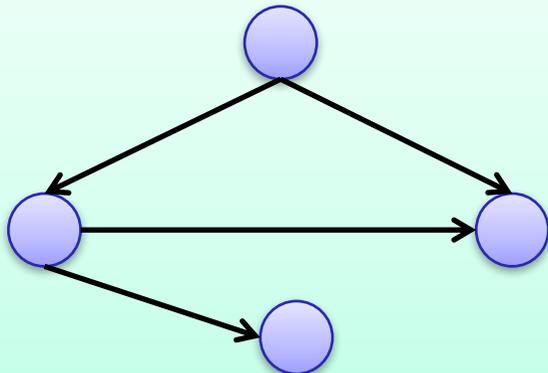
Tree of ESs



Easy to plug a new searcher by extending the Searcher class:

```
selectState() → ExecutionState  
update(addedStates, removedStates)
```

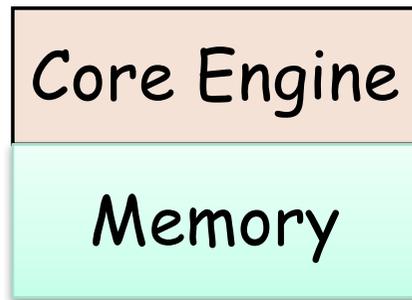
CFG



Statistics

- Solver time
- Instructions executed
- Memory consumption
- etc.

Memory Modelling



- One data type: **arrays of bitvectors (BVs)**
- 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
- We can translate all C expressions into constraints in the theory of quantifier-free BV with arrays (QF_ABV) with bit-level accuracy
 - Main exception: floating-point

Accuracy: Example

```
char buf[N]; // symbolic
```

```
struct pkt1 { char x, y, v, w; int z; } *pa = (struct pkt1*) buf;
```

```
struct pkt2 { unsigned i, j; } *pb = (struct pkt2*) buf;
```

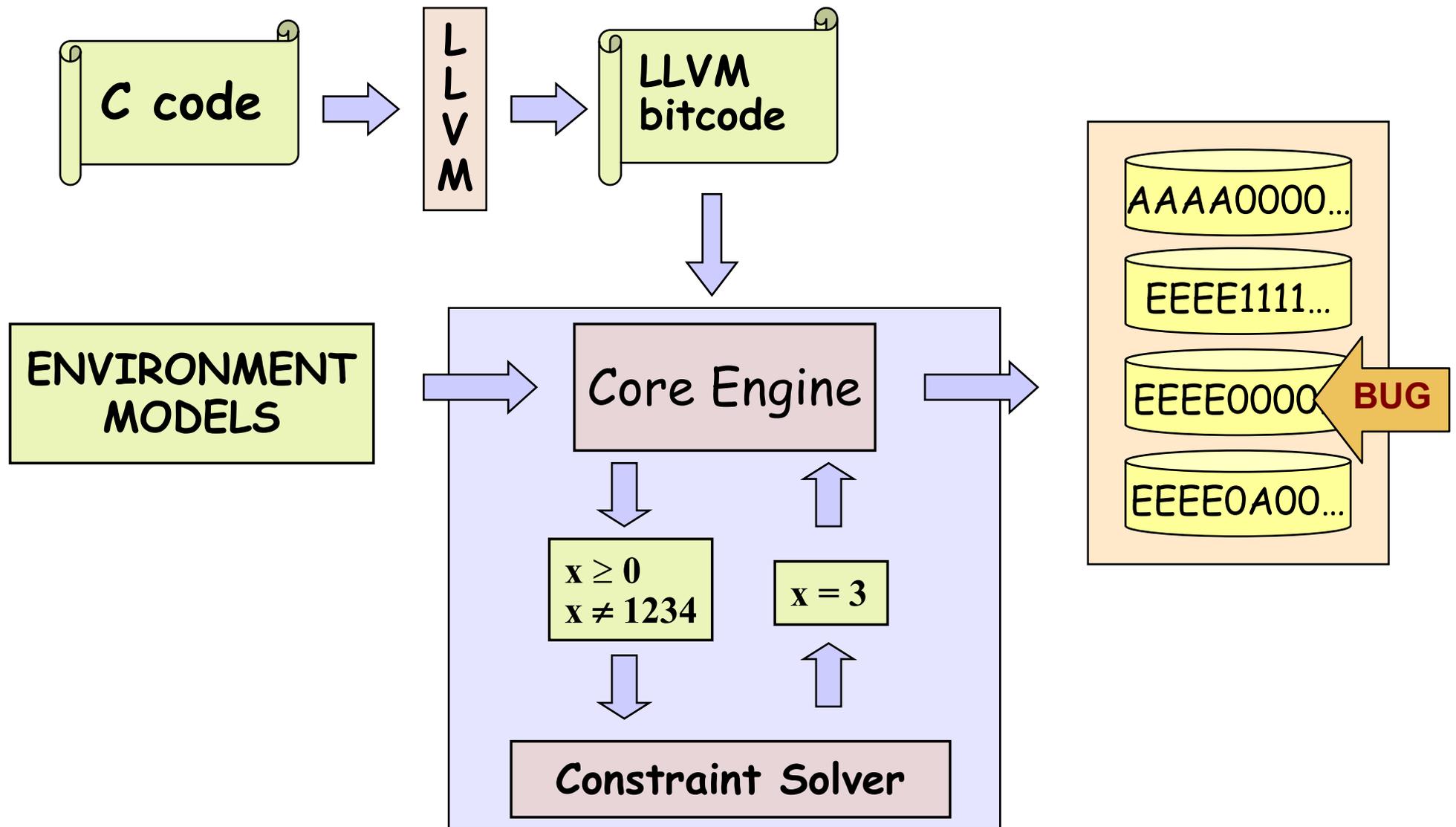
```
if (pa[2].v < 0) { assert(pb[2].i >= 1<<23); }
```

```
buf: ARRAY BITVECTOR(32) OF BITVECTOR(8)
```

```
buf[18] <SIGNED 0x00
```

```
buf[19]@buf[18]@buf[17]@buf[16] ≥UNSIGNED 0x00800000
```

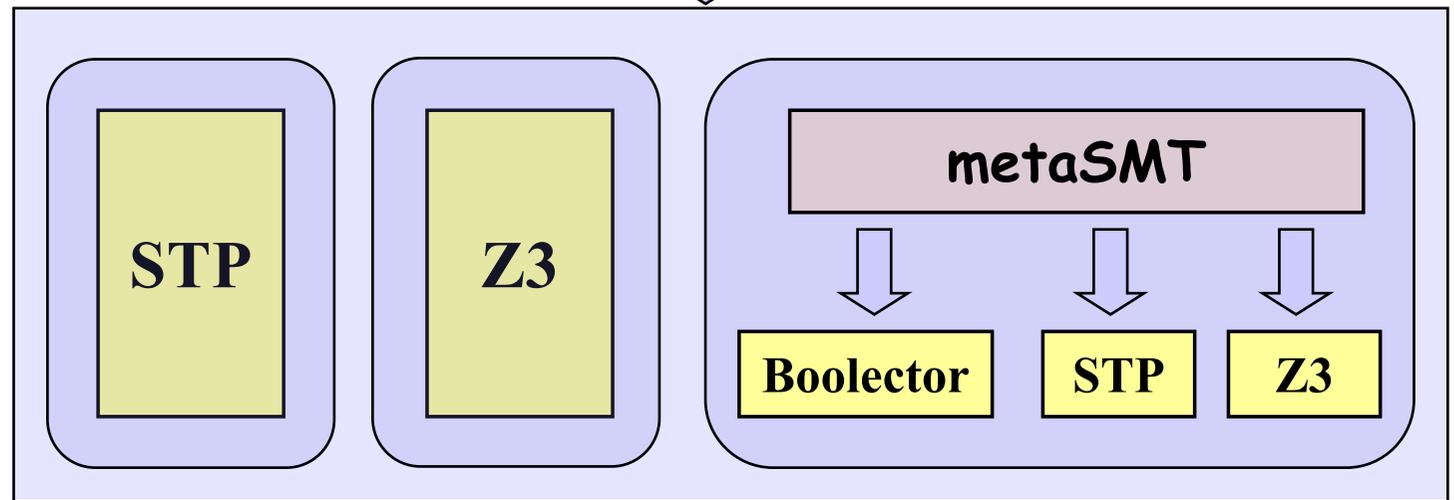
KLEE Architecture



SMT Solvers

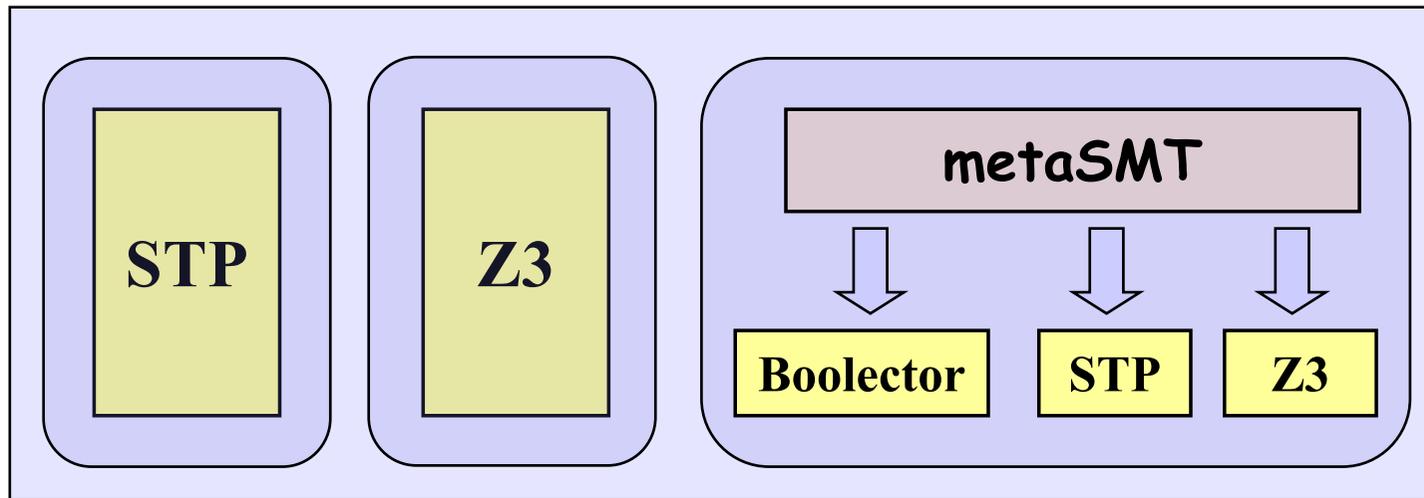
(`--solver-backend=stp, z3, ...`)

Theory of closed
quantifier-free
formulas over
bitvectors and
arrays of
bitvectors
(QF_ABV)



- **STP**: Developed at Stanford. Initially targeted to, and driven by, EXE. Main solver in KLEE.
- **Z3**: Developed at Microsoft Research, integrated both natively and as part of metaSMT.
- **Boolector**: Developed at Johannes Kepler University, integrated via metaSMT.

metaSMT

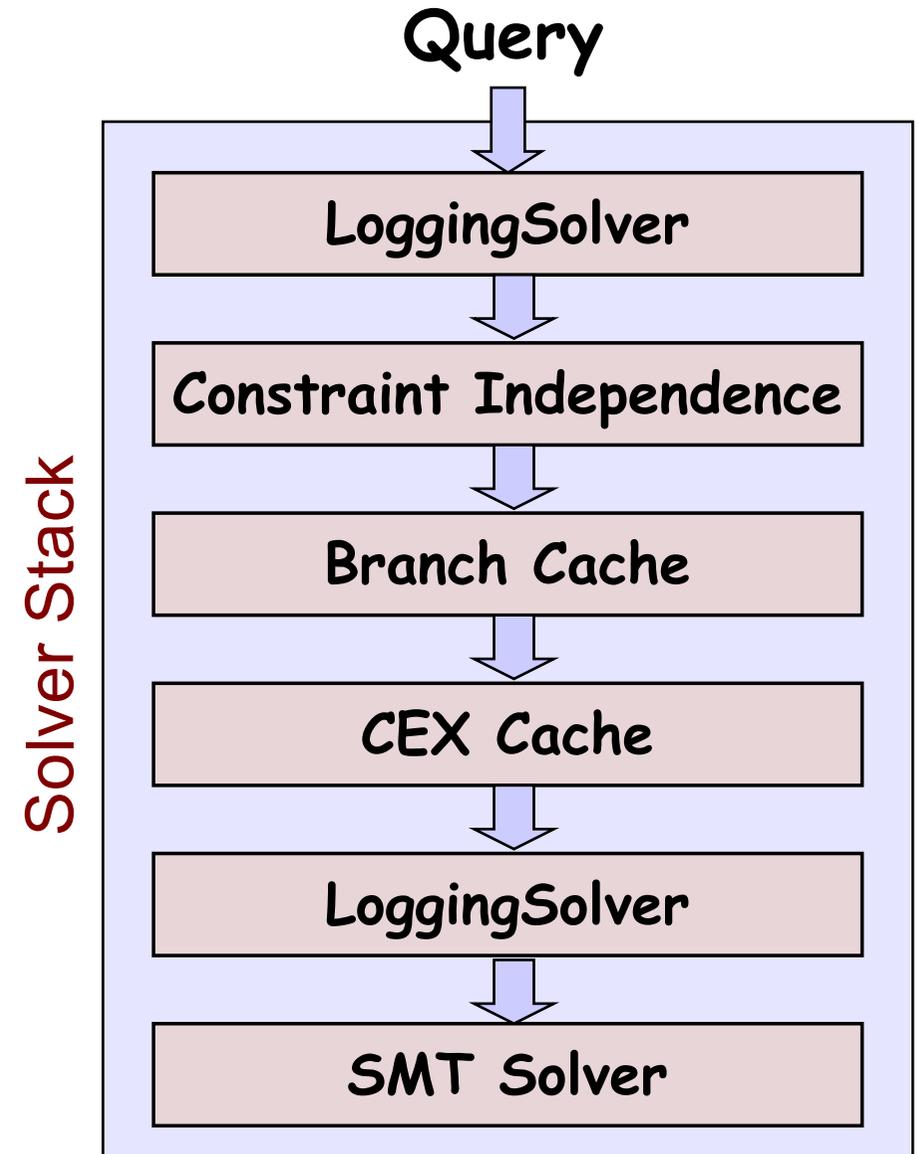


- **metaSMT** developed at University of Bremen provides a unified API for transparently using a number of SMT (and SAT) solvers
 - Avoids communication via text files, which would be too expensive
 - Small overhead: compile-time translation via metaprogramming

KLEE Architecture:

Constraint Solver

- Several high-level optimizations specific to symex
 - CEX caching, elimination of irrelevant constraints, etc.
- Implemented as a stack of solver passes
- Caching → only some queries reach the solver
- Independent **Kleaver** tool that implements this solver stack



Eliminating Irrelevant Constraints

(`--use-independent-solver=true/false`)

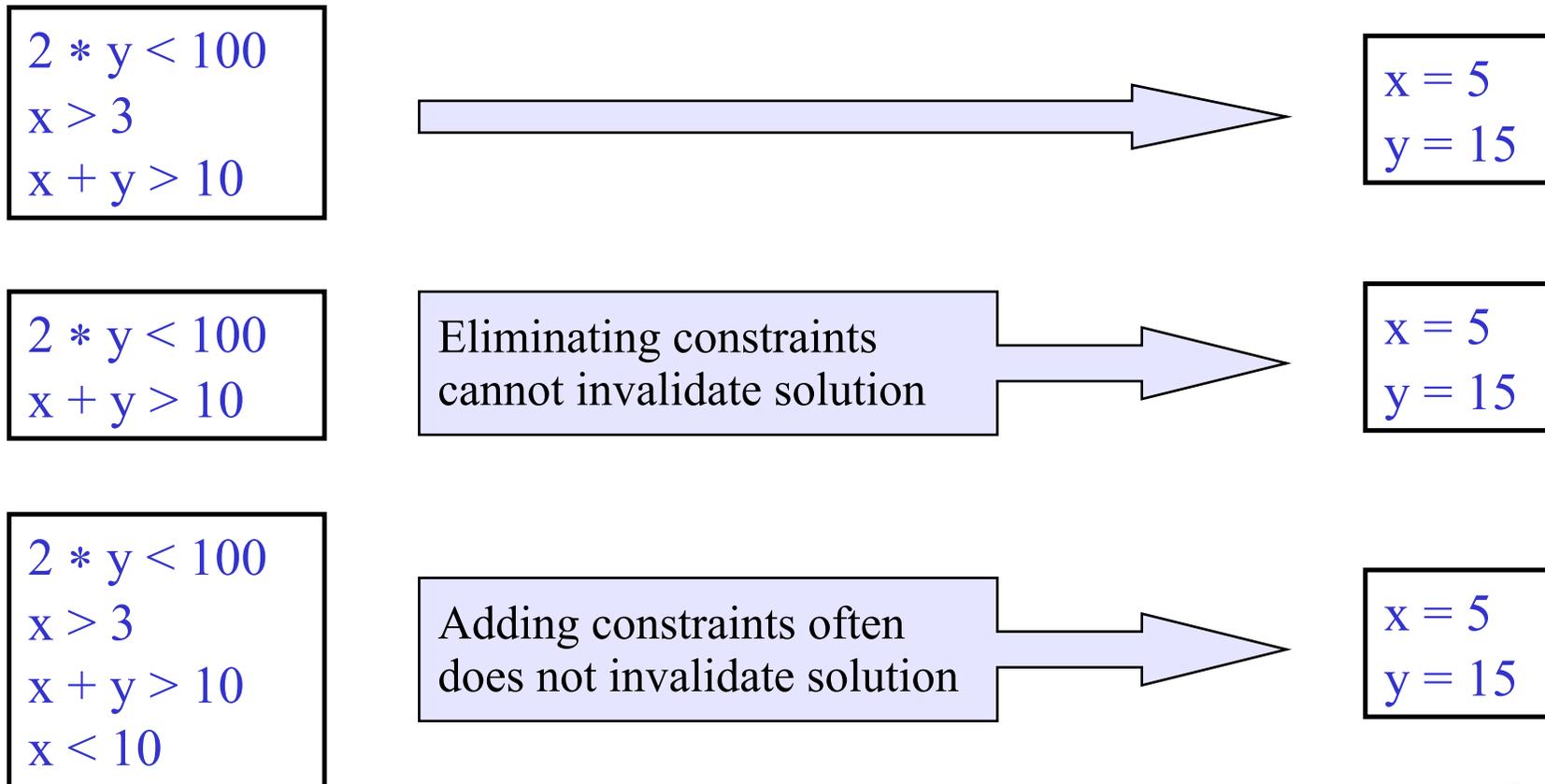
- In practice, each branch usually depends on a small number of variables

...	$w+z > 100$
...	$2 * w - 1 < 12345$
...	$x + y > 10$
...	$z \& z = z$
if (x < 10) {	$x < 10 ?$
...	
}	

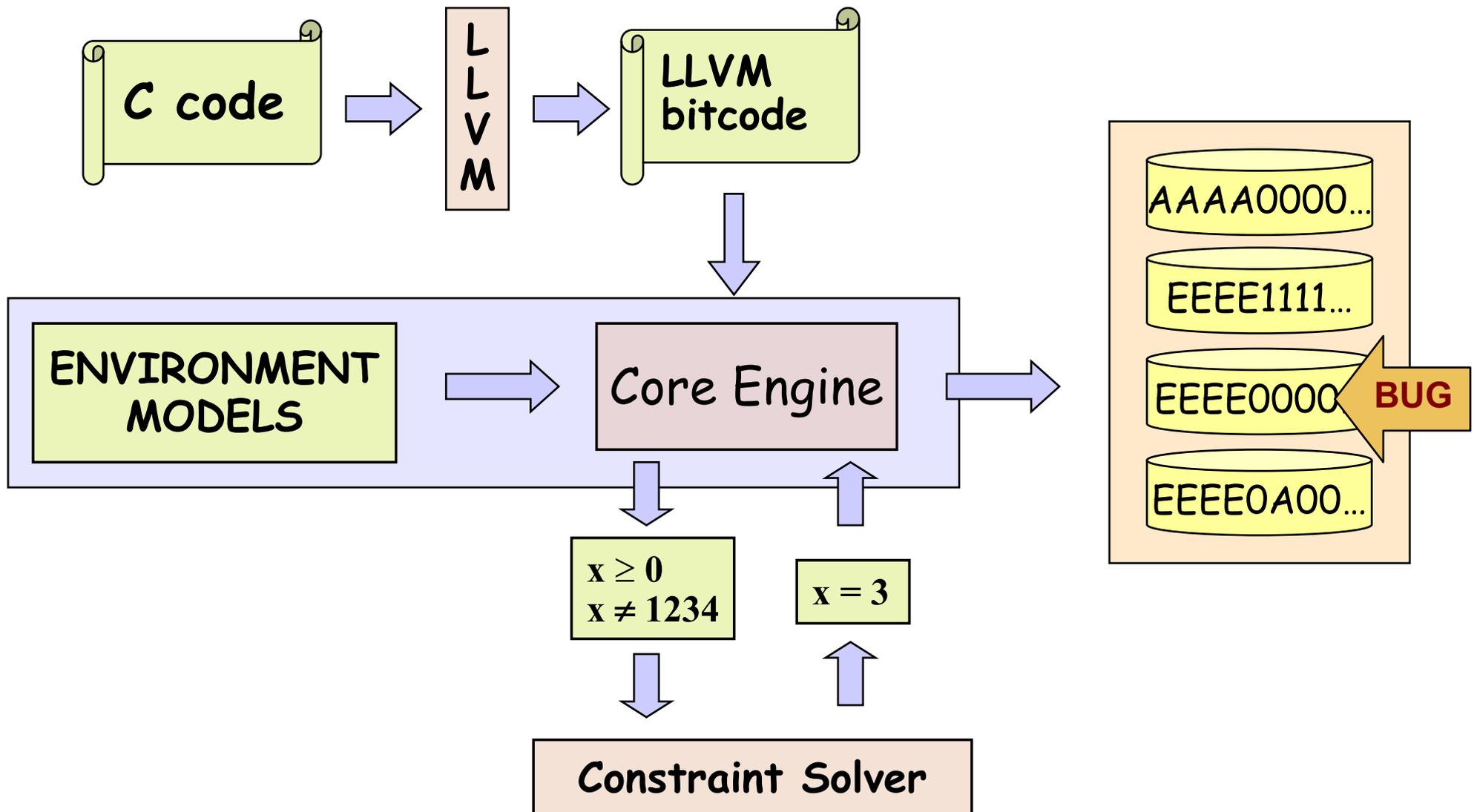
Caching Solutions

(`--use-cex-cache=true/false`)

- Static set of branches: lots of similar constraint sets



KLEE Architecture



KLEE Architecture:

Environment Models

-
- Environment model: model for a piece of code for which source is not available
 - In KLEE, the environment is mainly the OS system call API

Environmental Modeling

*Models are plain C code,
which KLEE interprets as
any other code!*

```
// actual implementation: ~50 LOC
ssize_t read(int fd, void *buf, size_t count) {
    klee_file_t *f = get_file(fd);
    ...
    memcpy(buf, f->contents + f->off, count)
    f->off += count;
    ...
}
```

- Users can extend/replace environment w/o any knowledge of KLEE's internals
 - Often the first part of KLEE users experiment with
- Users can choose precision
 - fail system calls? etc.
- Currently: effective support for symbolic command line arguments, files, links, pipes, ttys, environment vars

Statistics

Core Engine

Stats

The screenshot shows the klee-last/run.istats application interface. The top window title is "klee-last/run.istats [./echo.bc]". The interface includes a menu bar (File, View, Go, Settings, Help) and a toolbar with icons for Open, Reload, Force Dump, Up, Back, Forward, Show Relative Costs, Percentage Relative to Parent, and Do Cycle Detection. Below the toolbar is a "Flat Profile" window with a search field and a "(No Grouping)" dropdown. The profile table has columns for "Incl.", "Self", "Called", "Function", and "Location". The "Function" column is sorted by "Self" value in descending order. The "user_main" function is highlighted in blue. To the right is a "Source Code" window showing the C code for "echo.c". The code includes comments and function definitions. Line 138, "setlocale(LC_ALL, \"\");", is highlighted in blue, corresponding to the selected function in the Flat Profile. The code also shows calls to "setlocale" and "parse_long_options".

Incl.	Self	Called	Function	Location
300 673	1	(0)	main	assembly.ll
300 672	544	1	_uClibc_main	assembly.ll: _uCl
289 676	1 317	1	user_main	assembly.ll: echo.
220 591	43	1	setlocale	assembly.ll: locale
217 627	692	1	newlocale	assembly.ll: locale
107 771	8 522	2	_locale_set_l	assembly.ll: locale
98 085	98 085	19	memcpy	assembly.ll: mem
95 083	2 083	2	init_cur_collate	assembly.ll: locale
75 006	16 306	20	getenv	assembly.ll: geten
56 650	56 650	866	memcpy	assembly.ll: mem
48 510	18 756	6	find_locale	assembly.ll: locale
35 009	500	25	exit	assembly.ll: _atex
33 484	1 450	25	_exit_handler	assembly.ll: _atex
32 034	375	25	close_stdout	assembly.ll: closec
31 659	2 500	50	close_stream	assembly.ll: close-
26 788	5 095	50	fclose	assembly.ll: fclose
26 412	26 412	335	strncpy	assembly.ll: strnc
24 566	24 566	55	memset	assembly.ll: mem:
21 688	149	1	parse_long_options	assembly.ll: long-o
19 850	1 350	50	close	assembly.ll: fd.c
10 524	91	1	version_etc_va	assembly.ll: versic
10 026	59	1	rpl_vfprintf	assembly.ll: vfprin
9 720	664	1	vasprintf	assembly.ll: vaspr
8 362	15	1	_uClibc_init	assembly.ll: _uCl
7 555	37	1	snprintf	assembly.ll: snpri
7 518	101	1	vsprintf	assembly.ll: vsnpr
7 417	123	1	vfprintf	assembly.ll: vfpri
6 739	4	1	_locale_init	assembly.ll: locale
6 735	129	1	_locale_init_l	assembly.ll: locale
6 549	79	1	getopt_long	assembly.ll: getop
6 470	199	1	_getopt_internal	assembly.ll: getop
6 271	1 095	1	_getopt_internal_r	assembly.ll: getop
5 618	1 189	20	fwrite_unlocked	assembly.ll: fwrite
4 970	2 660	35	_stdio_WRITE	assembly.ll: _WRI
4 894	236	1	_ppfs_init	assembly.ll: _vfpri
4 611	4 611	62	strlen	assembly.ll: strlen

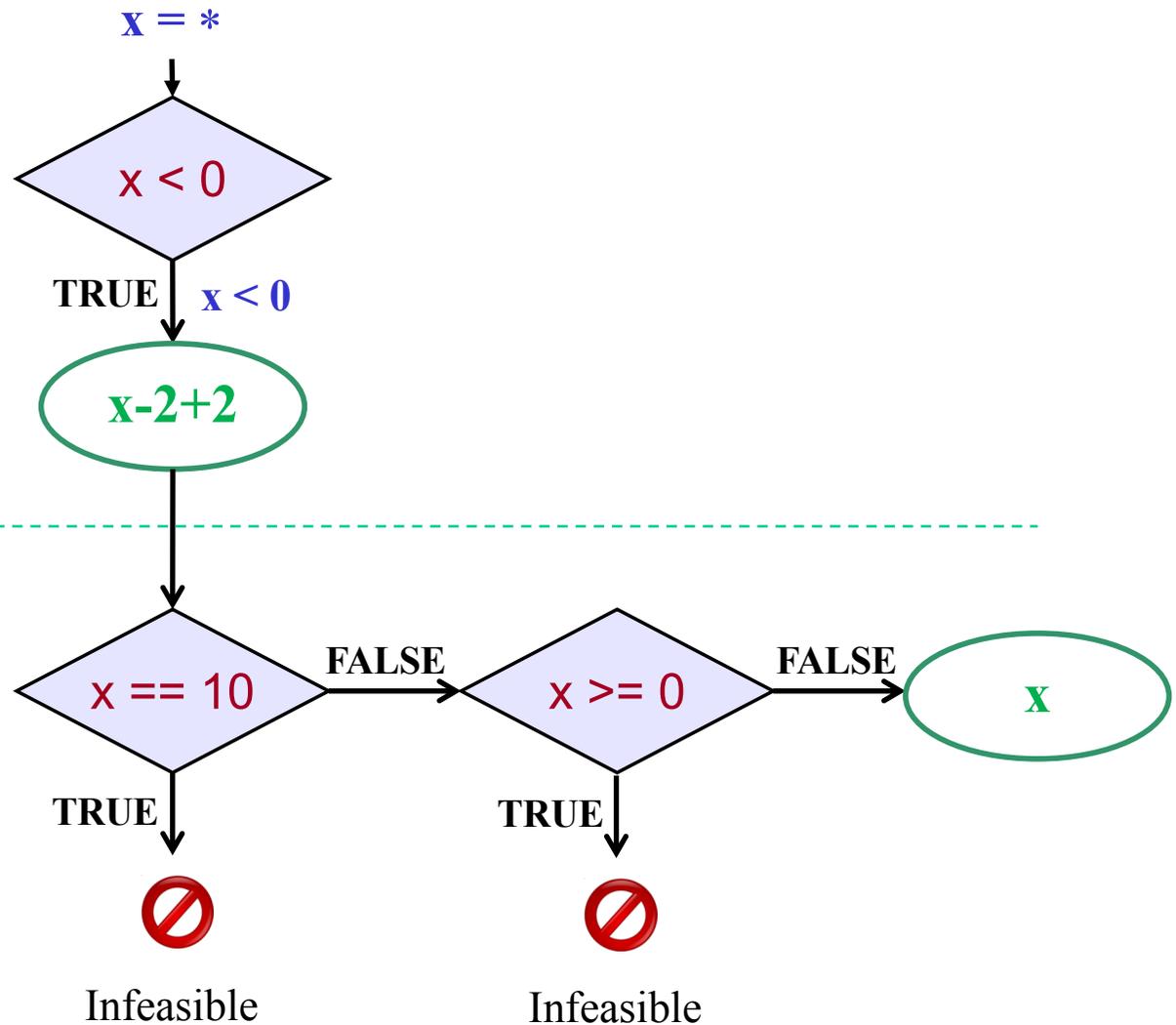
Good support for producing and visualizing a variety of statistics, associated with different entities and events

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;
```

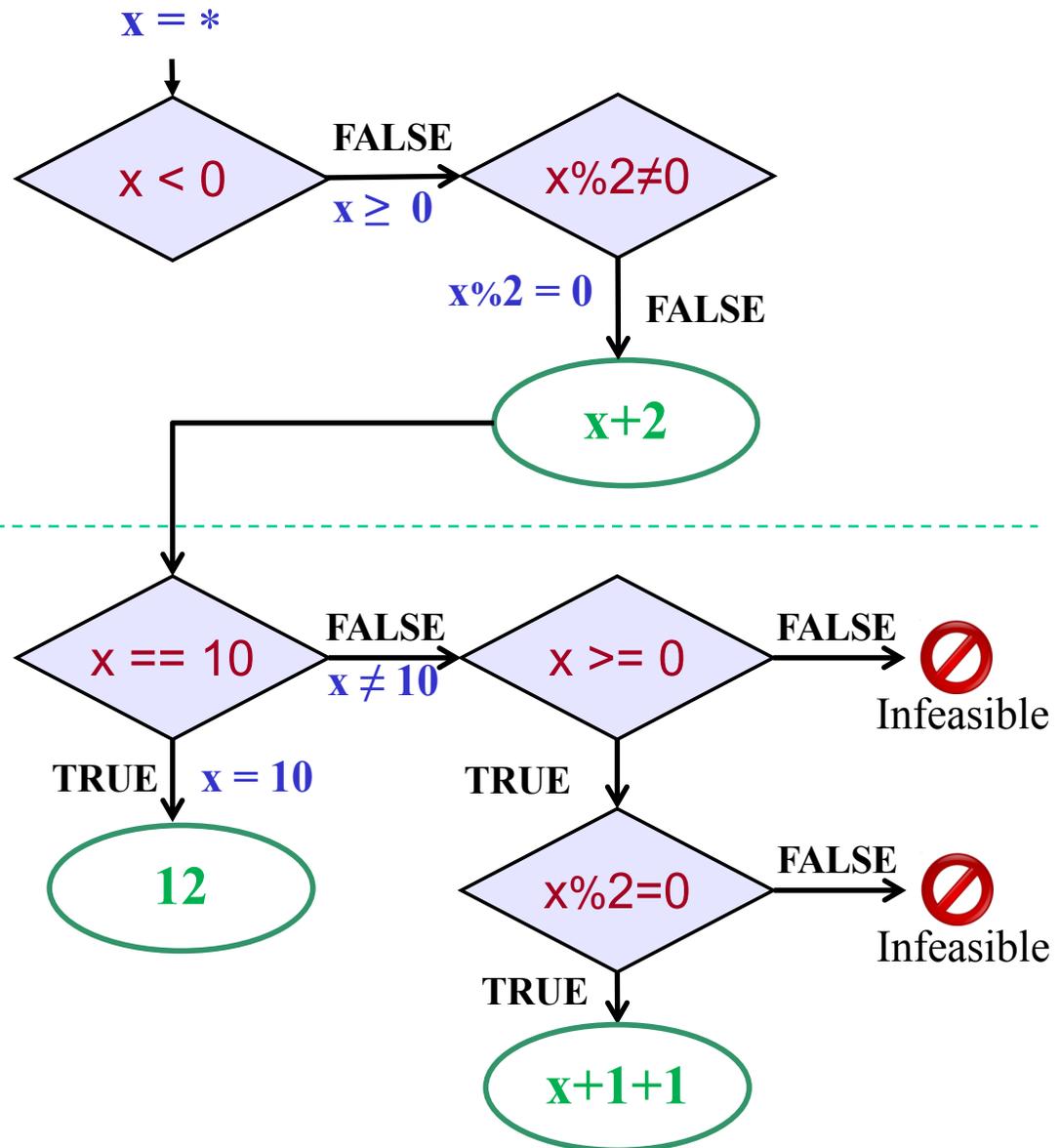


Crosschecking Two Software Versions

```
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;
```



KLEE: Freely Available as Open-Source

<http://klee.github.io/>

- Popular symbolic execution tool with an active user and developer base
- Extended in many interesting ways by several groups from academia and industry, in areas such as:
 - exploit generation
 - wireless sensor networks/distributed systems
 - automated debugging
 - client-behavior verification in online gaming
 - GPU testing and verification
 - etc. etc.