Symbolic Execution in EXE / KLEE

Cristian Cadar, Daniel Dunbar, Dawson Engler

STANFORD
UNIVERSITY

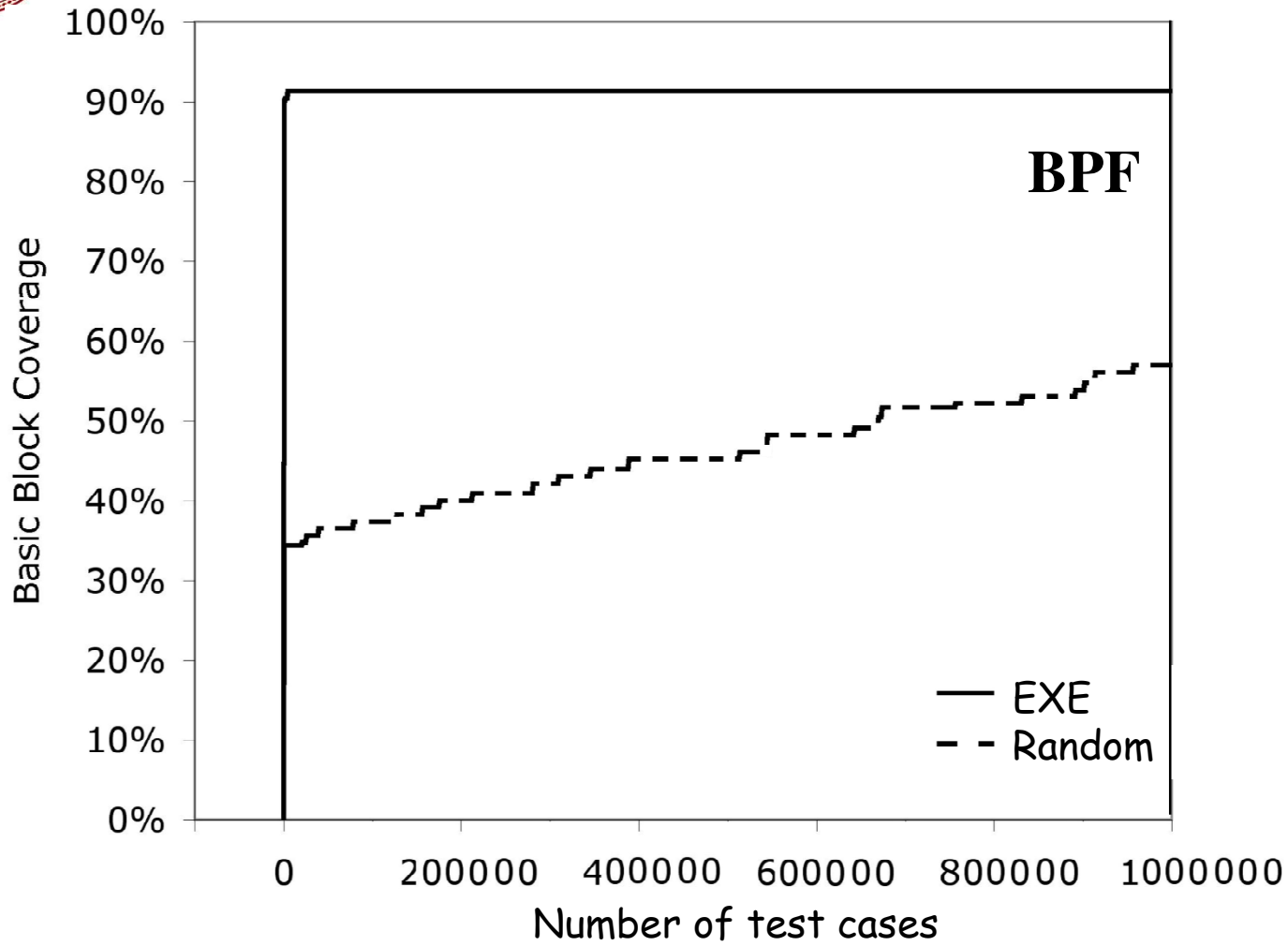
CS.295 - May 28th, 2008

Motivation

- Testing is hard
  - Manual testing is very expensive
  - Random ("fuzz") testing is often ineffective
    - Hard to hit narrow input ranges
    - Hard to generate structured input (e.g. compiler)

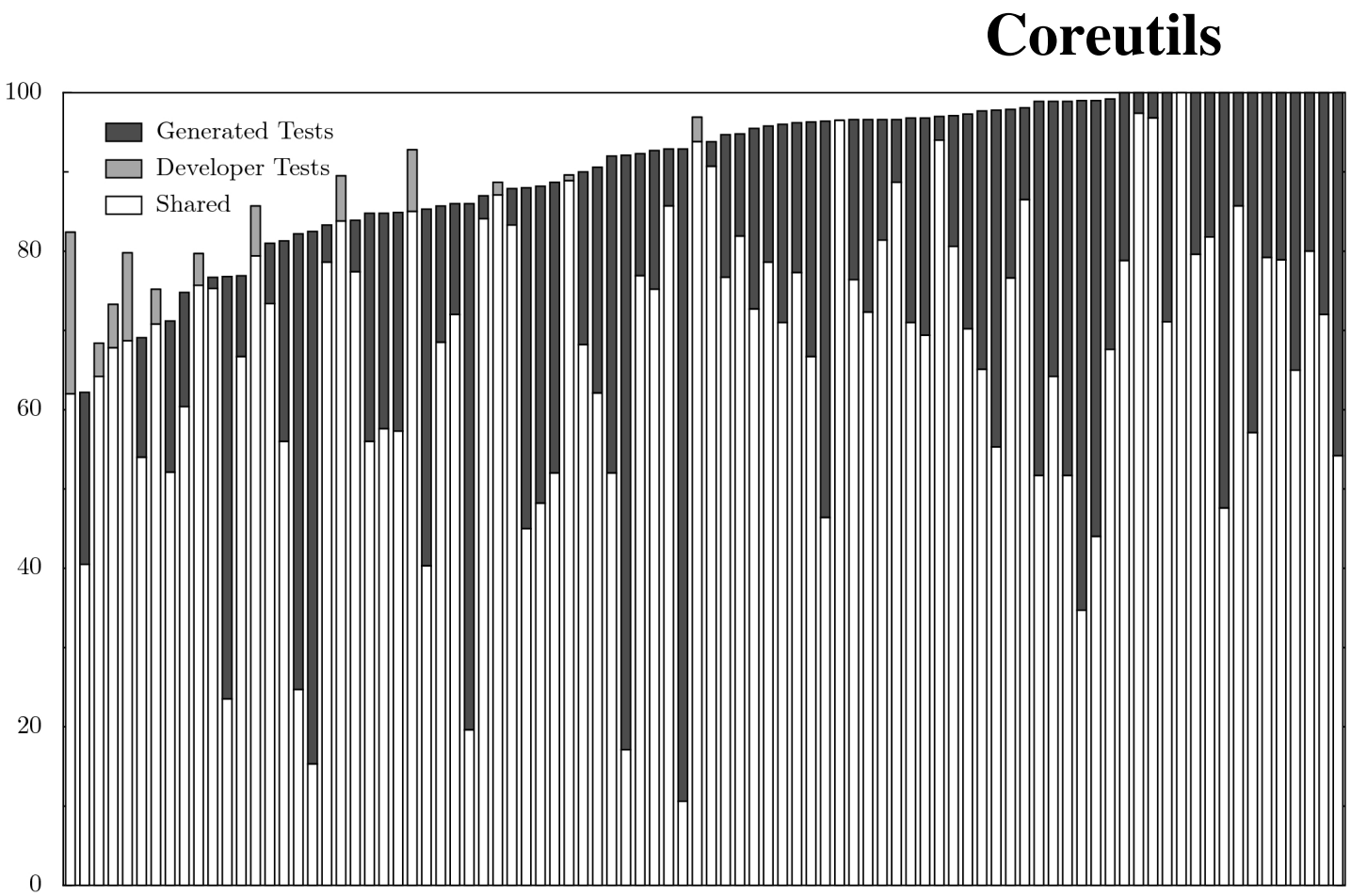
```
int bad_abs(int x) {
    if(x < 0)
        return -x;
    if(x == 12345678)
        return -x;
    return x;
}
```

Sym Ex vs. Random Testing



[EXE: *Automatically generating inputs of death*
C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, D. Engler, CCS 2006]

Sym Ex vs. Manual Testing



Sym ex vs dyn/static analysis

Sym ex vs dyn/static analysis

- Dynamic analysis requires test cases
- Static analysis is imprecise
  - – hard to find bugs dependent on specific values and/or memory layout; functional bugs (e.g. crosschecking)
  - – false positives
  - – does not generate test cases
  - + but it usually finds more bugs
  - + easier to apply (don't need full program)
- Both are complementary
  - – No reason not to run static; can use static info to improve symbolic execution
  - – Can run tests cases generated by symbolic execution on dynamic tools

Symbolic Execution

- Automatic
  - does not require test cases
- Highly systematic
  - reaches deep code paths
  - achieves high statement/branch coverage
  - can check error paths
- Example:
  - Coreutils overall statement coverage: 77.2%
  - Coreutils apps at 100%: 16/90
  - Coreutils apps over 90%: 54/90, over 80%: 70/90

Symbolic Execution (2)

- Finds bugs
  - including those depending on specific values and/or memory layout
  - including functional bugs (see crosschecking study)
- Generates concrete test cases for explored paths
  - error reports for paths hitting a bug
- Example: crashing ext3 on Linux 2.6.10

Disk of death (JFS, Linux 2.6.10)

Offset	*Hex Values*
00000	0000 0000 0000 0000 0000 0000 0000 0000
. . .	. . .
08000	464a 3153 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 0000 0000 0000 0000 0000
. . .	. . .
10000	

[*Automatically generating malicious disks using symbolic execution*
J. Yang, C. Sar, P. Twohey, C. Cadar, D. Engler , IEEE Security 2006]

Basic idea

- Run program on symbolic input, whose initial value is *anything*
- Program instructions become operations on symbolic expressions
- At conditionals that use symbolic inputs, fork execution and follow both paths:
  - On true branch, add constraint that condition is true
  - On false, that it is not
- When a path terminates, generate a test case by solving the constraints on that path

Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
        return skb->data + offset;
    exit(1);
}
…
offset = read_from_user();
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
        return skb->data + offset;
    exit(1);
}
…
exe_make_symbolic(&offset, sizeof(offset));
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
        return skb->data + offset;
    exit(1);
}
…
exe_make_symbolic(&offset, sizeof(offset));
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
            return skb->data + offset;
    exit(1);
}
…
exe_make_symbolic(&offset, sizeof(offset));
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
        return skb->data + offset;
    exit(1);
}
…
exe_make_symbolic(&offset, sizeof(offset));
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
        return skb->data + offset;
    exit(1);
}
…
exe_make_symbolic(&offset, sizeof(offset));
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

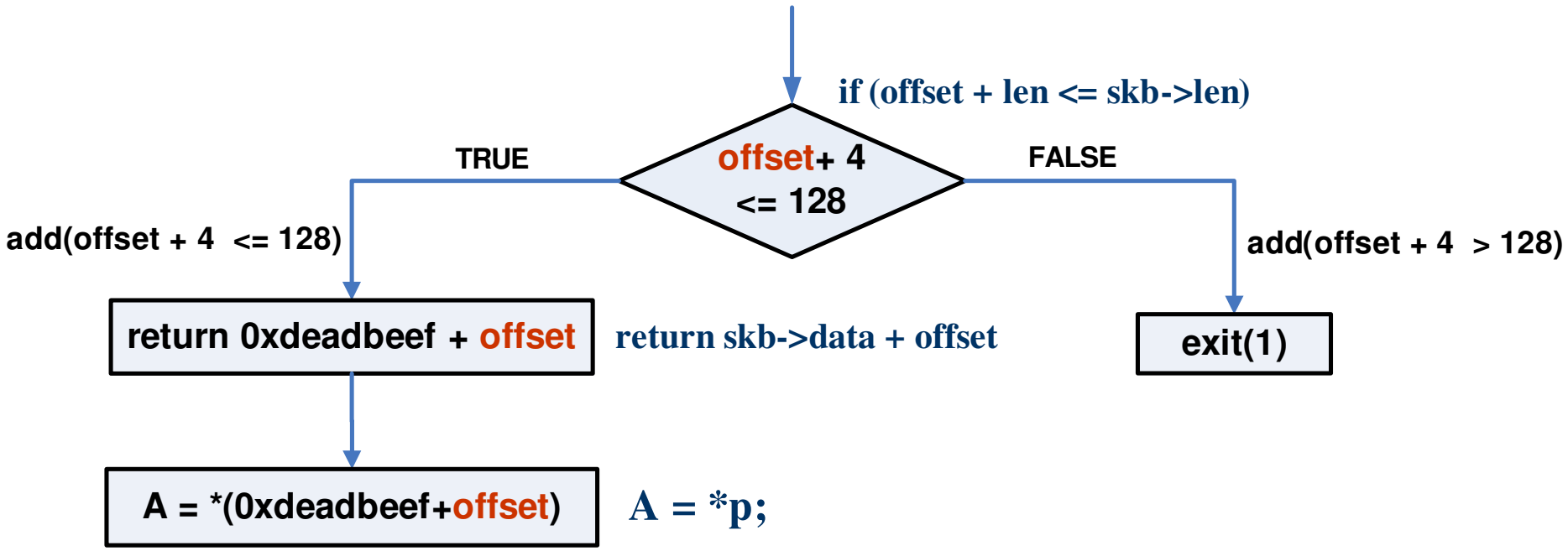
Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
        return skb->data + offset;
    exit(1);
}
…
exe_make_symbolic(&offset, sizeof(offset));
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

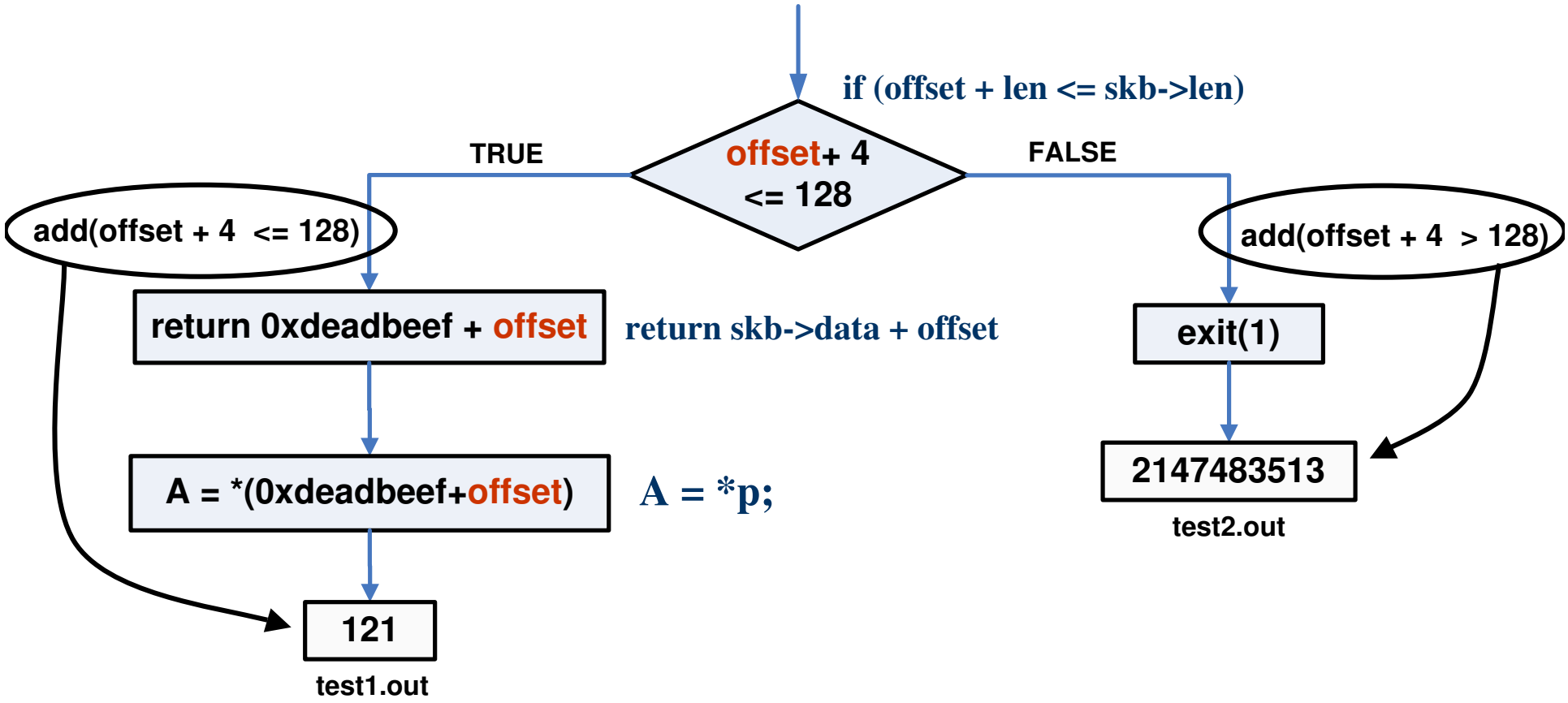
Example (simplified BPF code)

```
static inline void *skb_header_pointer(struct sk_buff  *skb,
                                       int offset,
                                       int len) {
    if (offset + len <= skb->len)
            return skb->data + offset;
    exit(1);
}
…
exe_make_symbolic(&offset, sizeof(offset));
…
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```

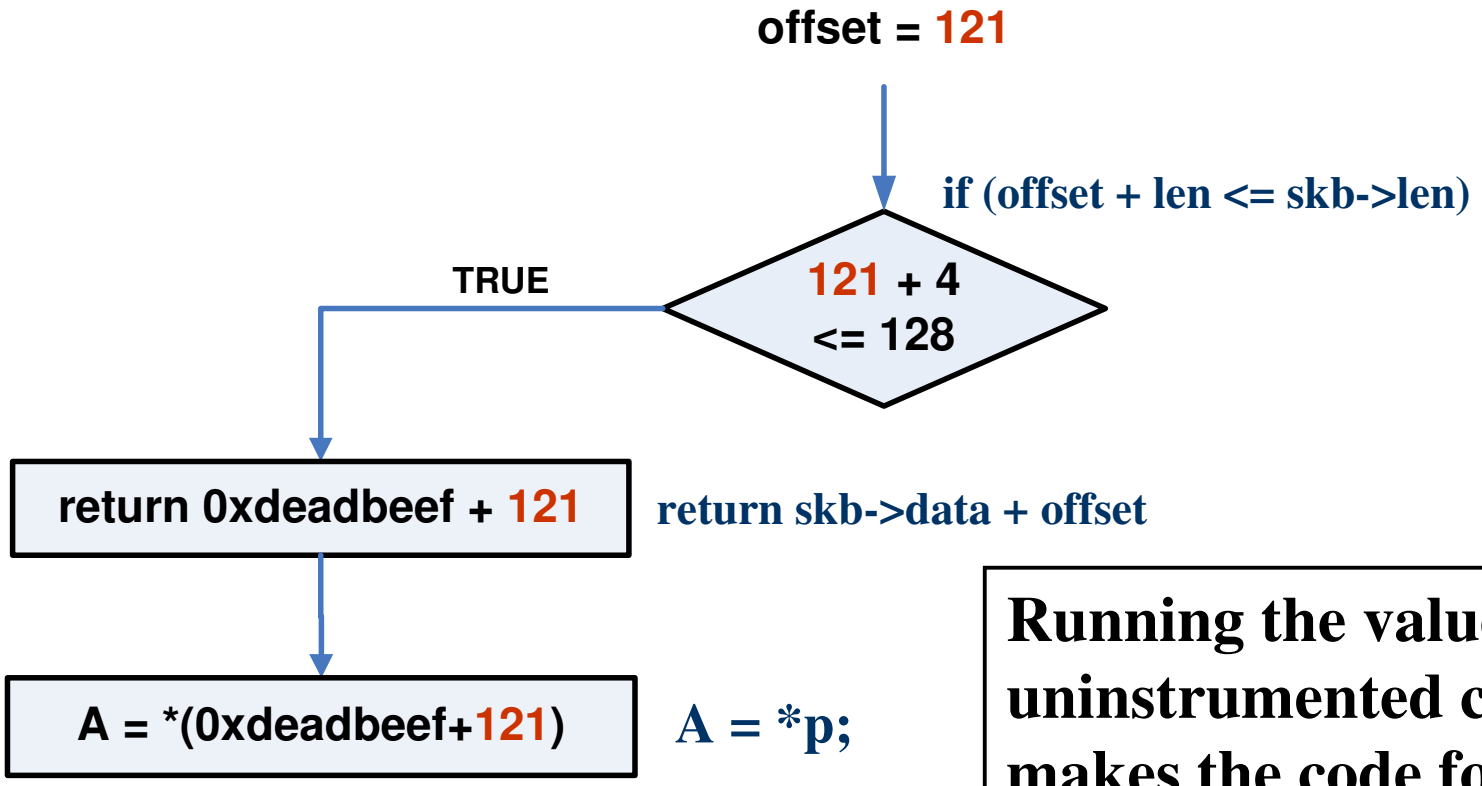
EXE execution



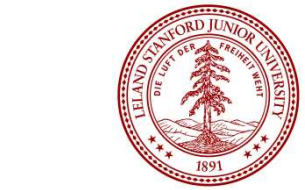
EXE execution



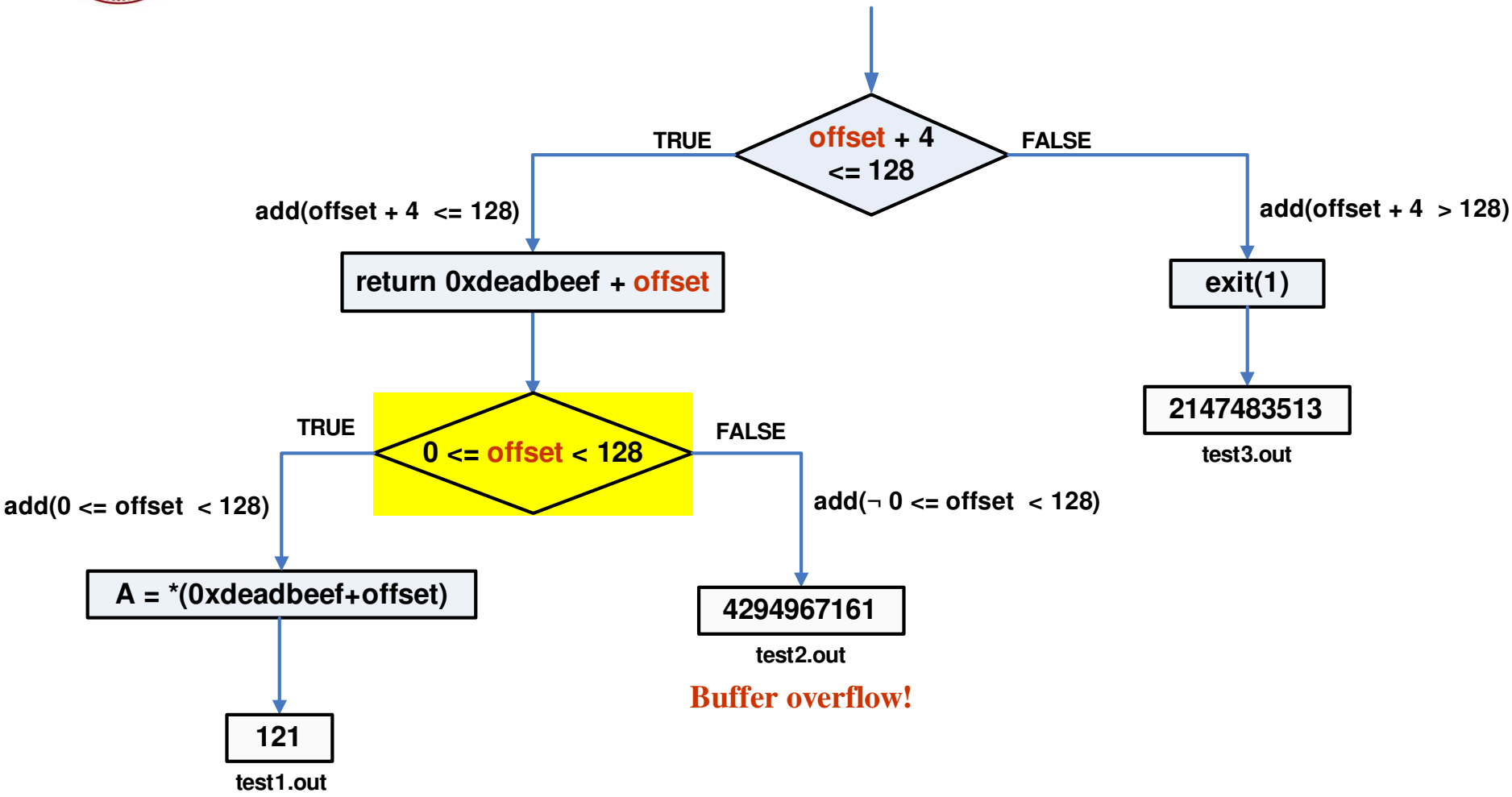
EXE execution



Running the values on the uninstrumented code makes the code follow the exact path on which the values were generated.



Implicit checks



Checks

- Default checks
  - Memory errors
    - Buffer overflows, NULL dereferences, double frees
  - Division/modulo by zero
  - Assertion violations
- Easy to extend EXE with arbitrary checks
  - Usually, asserts suffice:

```
assert(compress(uncompress(x)) == x);
```

Environment Problem

- Many programs are controlled by environment input
  - e.g., command-line arguments, files, environment variables, keyboard, network
- Would like to explore all interactions with environment
- Code is not available

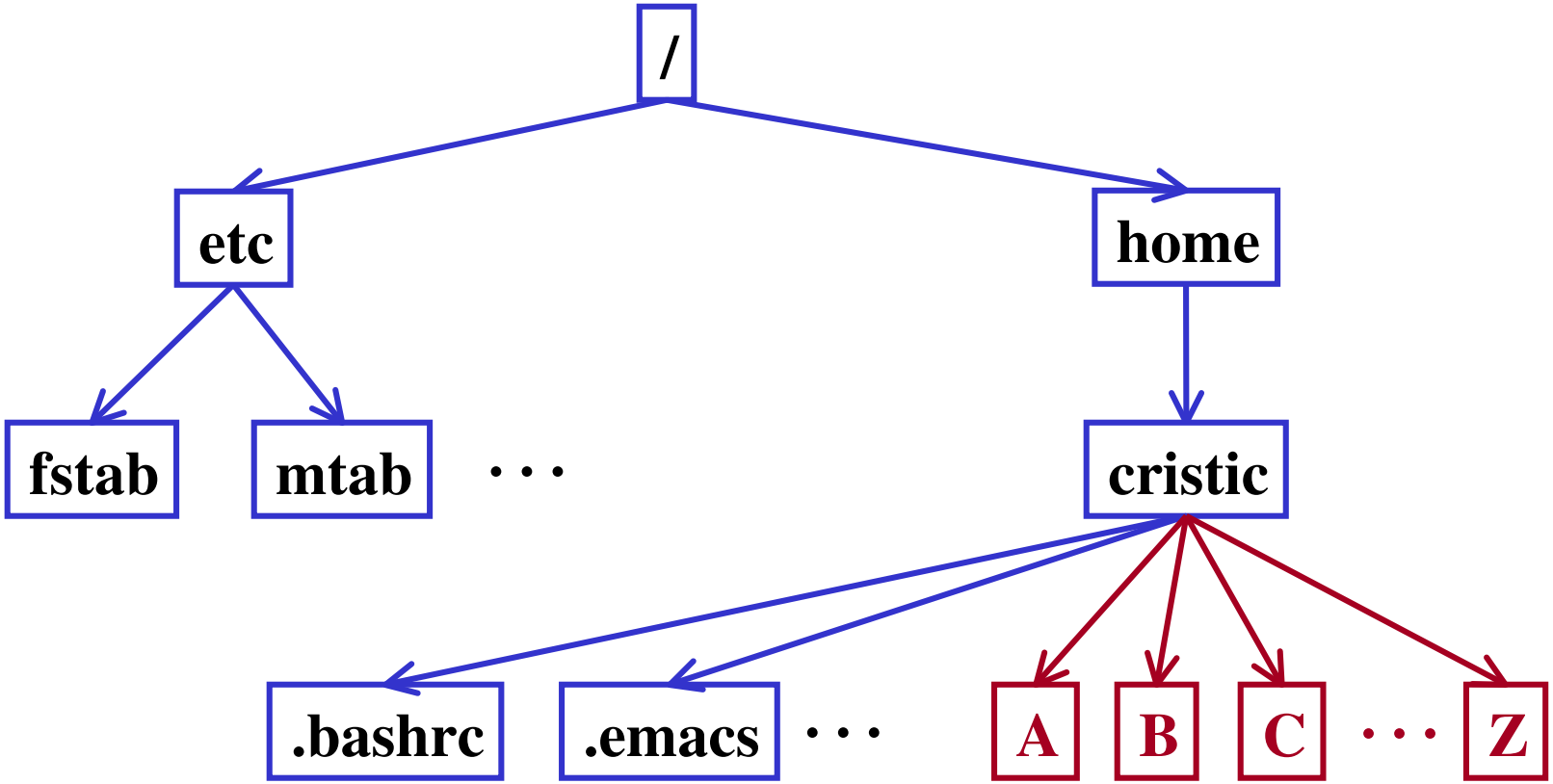
Environment Problem

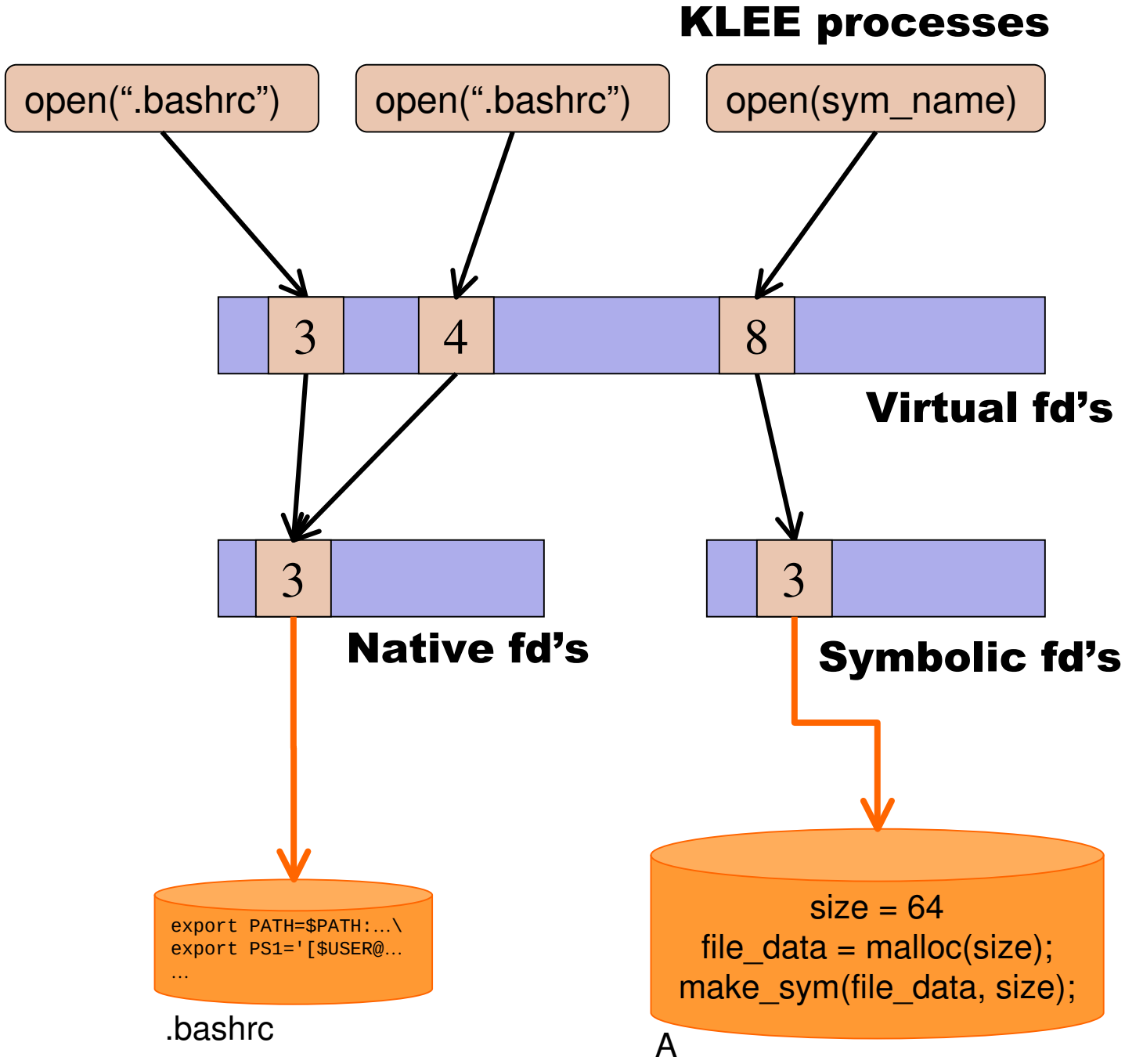
- *Option 1*: Choose concrete parameters before calling into environment
  - Problems: (1) missing paths, (2) interaction between EXE processes (why?)
- *Options 2*: Make results from environment unconstrained
  - e.g. , c = getchar() make_symbolic(&c, 1);
  - Problems: (1) tedious, (2) invalid paths

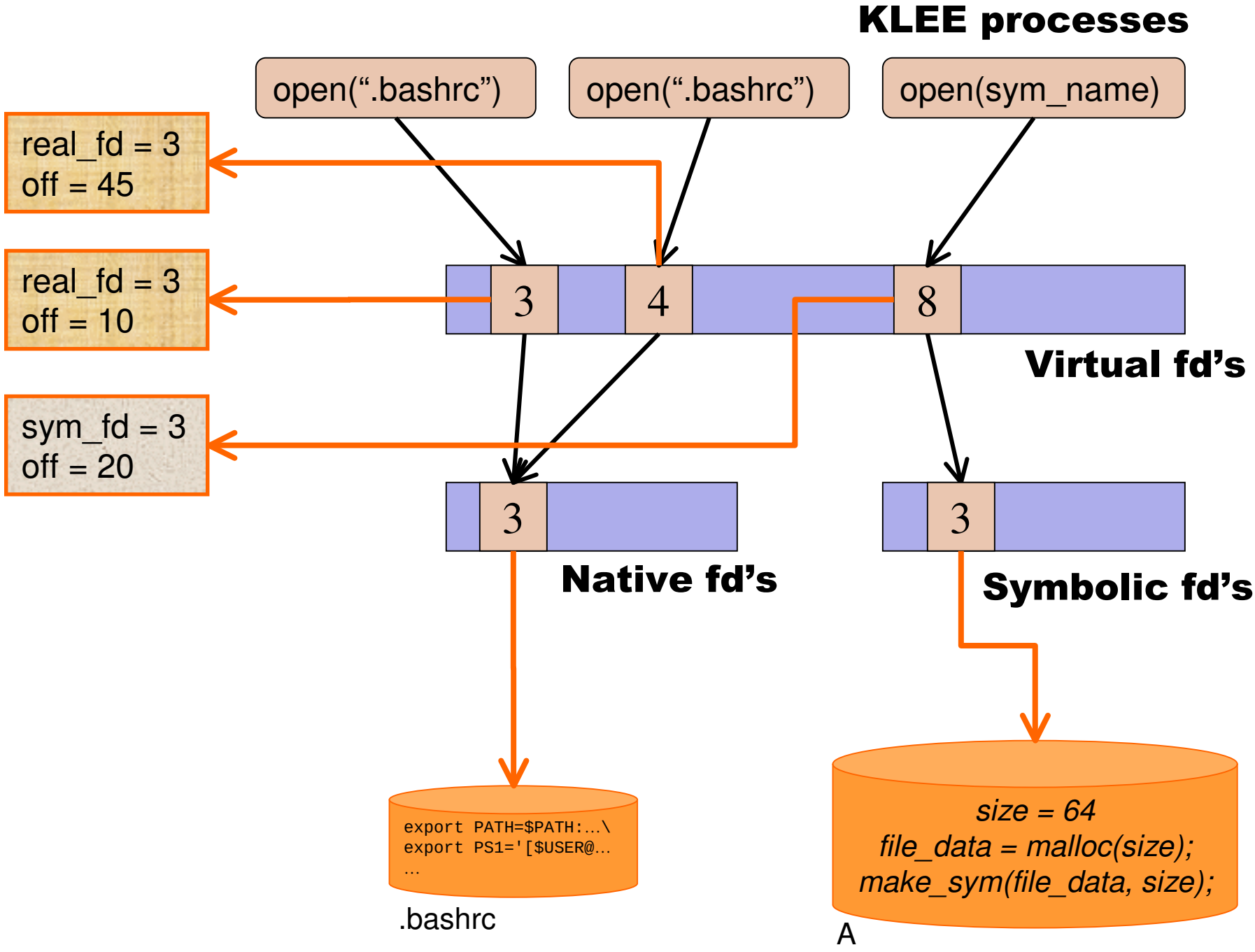
Symbolic Environment

- Provide a symbolic file system, symbolic loader etc.
  - virtualizes access to native resources among KLEE processes
  - provides symbolic arguments, files with symbolic contents etc.
  - implemented at the OS level (POSIX layer) (why?)
- Additional advantage:
  - No make_symbolic calls, just simple parameters (# command-line arguments, #symbolic files) to KLEE

Symbolic File System




KLEE processes
open(“.bashrc”)
open(“.bashrc”)
open(sym_name)
3
4
8
Virtual fd’s
3
3
Native fd’s
Symbolic fd’s
export PATH=$PATH:...\
export PS1='[$USER@...
...
.bashrc
size = 64
file_data = malloc(size);
make_sym(file_data, size);
A


KLEE processes
open(“.bashrc”)
open(“.bashrc”)
open(sym_name)
real_fd = 3
off = 45
real_fd = 3
off = 10
sym_fd = 3
off = 20
3
4
8
Virtual fd’s
3
Native fd’s
3
Symbolic fd’s
export PATH=$PATH:...\
export PS1='[$USER@...
...
.bashrc
size = 64
file_data = malloc(size);
make_sym(file_data, size);
A


LELAND STANFORD JUNIOR UNIVERSITY
DIE LUFT DER FREIHEIT WEHT
1891

Demo