IEEE SecDev | 2020

IEEE COMPUTER SOCIETY

Tutorial: Detecting Memory Vulnerabilities in the Components of System Code using PROMPT

Tuba Yavuz and Ken (Yihang) Bai

tuba@ece.ufl.edu, baiyihang@ufl.edu

University of Florida

#IEEESecDev https://secdev.ieee.org/2020

Tutorial Outline

- Part 0: Important information about the tool
- Part I: Background on Symbolic Execution and KLEE
- Part II: PROMPT for API Model Guided Symbolic Execution
- Part III: Real-World Case Studies
- Acknowledgements
- Q & A

Important information

- PROMPT github page:
  - https://github.com/sysrel/PROMPT
  - Manual: https://github.com/sysrel/PROMPT/blob/master/docs/PROMPT_Manual.txt
  - PROSE template:
    - https://github.com/sysrel/PROMPT/blob/master/docs/PROSE_template.txt
  - Tutorial examples:
    - https://github.com/sysrel/PROMPT/tree/master/PROMPT_examples
  - Device driver and BlueZ case studies
    - https://github.com/sysrel/PROMPT/tree/master/JASE_benchmarks
  - Heartbleed
    - https://github.com/sysrel/PROMPT/tree/master/caseStudies/openssl/heartbleed

Important information

- PROMPT virtual machine image for VirtualBox (Ubuntu 16.04, 64 bit)
  - https://drive.google.com/file/d/1bjLJ_HVCWBuQQC_6BNER3oiwEkXJGIHR/view?usp=sharing
  - Username: prompt passwd: prompt
  - VirtualBox can be downloaded from https://www.virtualbox.org/wiki/Downloads
- PROMPT source tree can be found at /home/prompt/PROMPT
- PROMPT has been built under /home/prompt/prompt_build_dir
- Set the PROMPT environment variable to the path of the executable
  - $ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
- Each tutorial example and real-world case study directory has a run.sh script
  - Follow the instructions on the slides to execute PROMPT using run.sh

PART I: Background on Symbolic Execution and KLEE

Outline for PART I

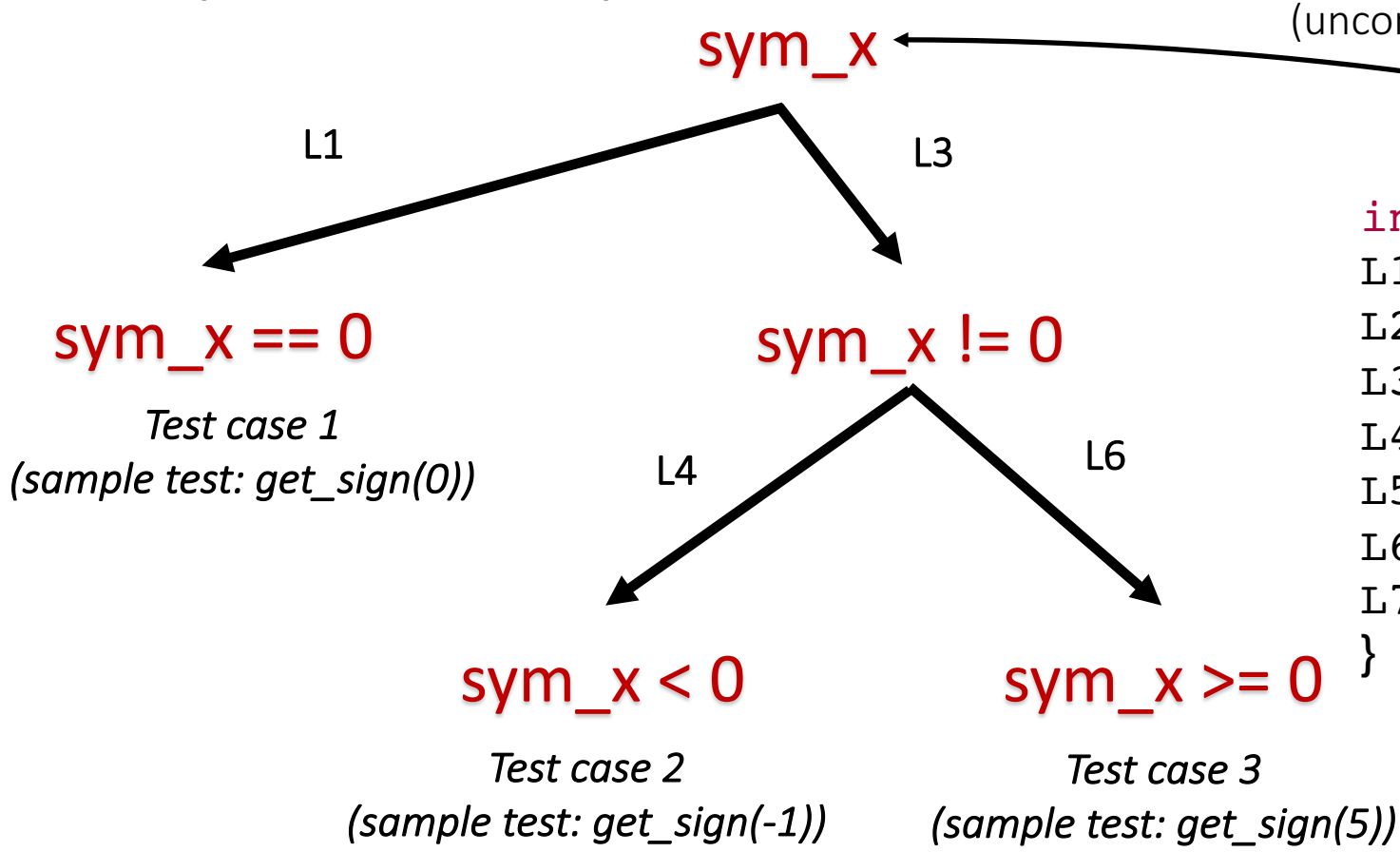
- What is symbolic execution?
- A simple example
- The KLEE symbolic execution engine
- Challenges of symbolic execution

What is Symbolic Execution?

- Symbolic execution is a program analysis technique that has been introduced as a testing technique by James King:
  - James C. King. Symbolic Execution and Program Testing. Communications of the ACM, 19:7, 1976.
- Symbolic execution interprets a program by propagating symbolic inputs according to the program logic
- There are two major approaches to symbolic execution
  - Concolic execution
  - Execution Generation Based Testing (EGT)
    - KLEE follows the EGT approach

A simple example

symbolic value of x at the time of calling get_sign
(unconstrained or can be anything)




```
int get_sign(int x) {
L1:  if (x == 0)
L2:    return 0;
L3:
L4:   if (x < 0)
L5:     return -1;
L6:   else
L7:     return 1;
}
```

100% Instruction/Code Coverage

100% Branch Coverage

KLEE Symbolic Execution Engine

- Very mature tool
  - Cristian Cadar, Daniel Dunbar, Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008.
  - Authors received ACM SIGOPS Hall of Fame Award due to impact of the paper
- Open source: https://github.com/klee/
  - Documentation: http://klee.github.io/
- Based on the LLVM IR
  - Code needs to be compiled with the clang compiler to generate the LLVM bitcode
- Follows the EGT approach
- Works on closed programs
  - Needs a test driver for proper initialization of data

A test driver for KLEE

```
#include “klee/klee.h”

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

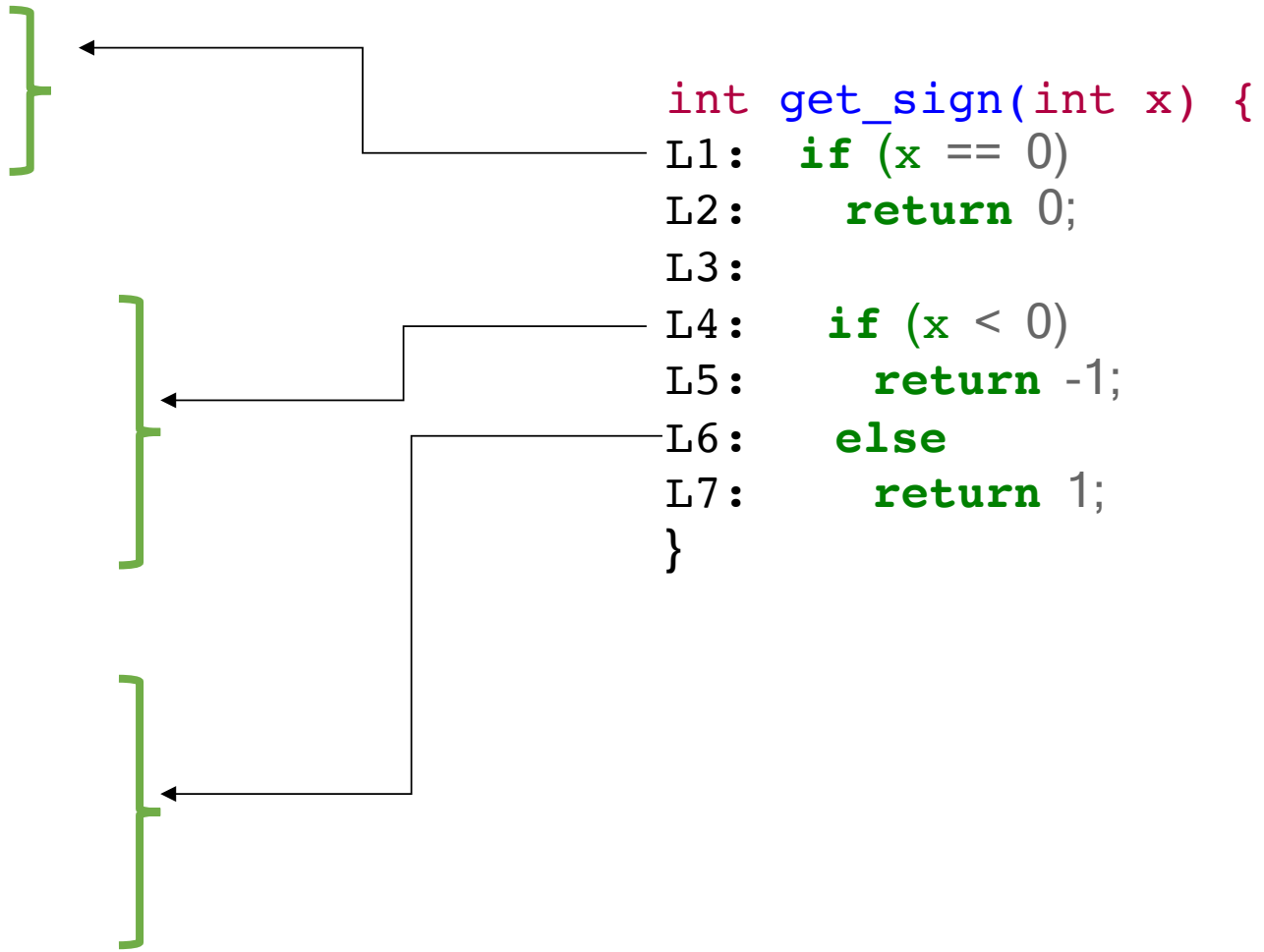
```
int get_sign(int x) {
L1:  if (x == 0)
L2:    return 0;
L3:
L4:   if (x < 0)
L5:     return -1;
L6:   else
L7:     return 1;
}
```

$ clang **-I** $KLEE_INCLUDE_PATH **-emit-llvm -c -g** get_sign.c

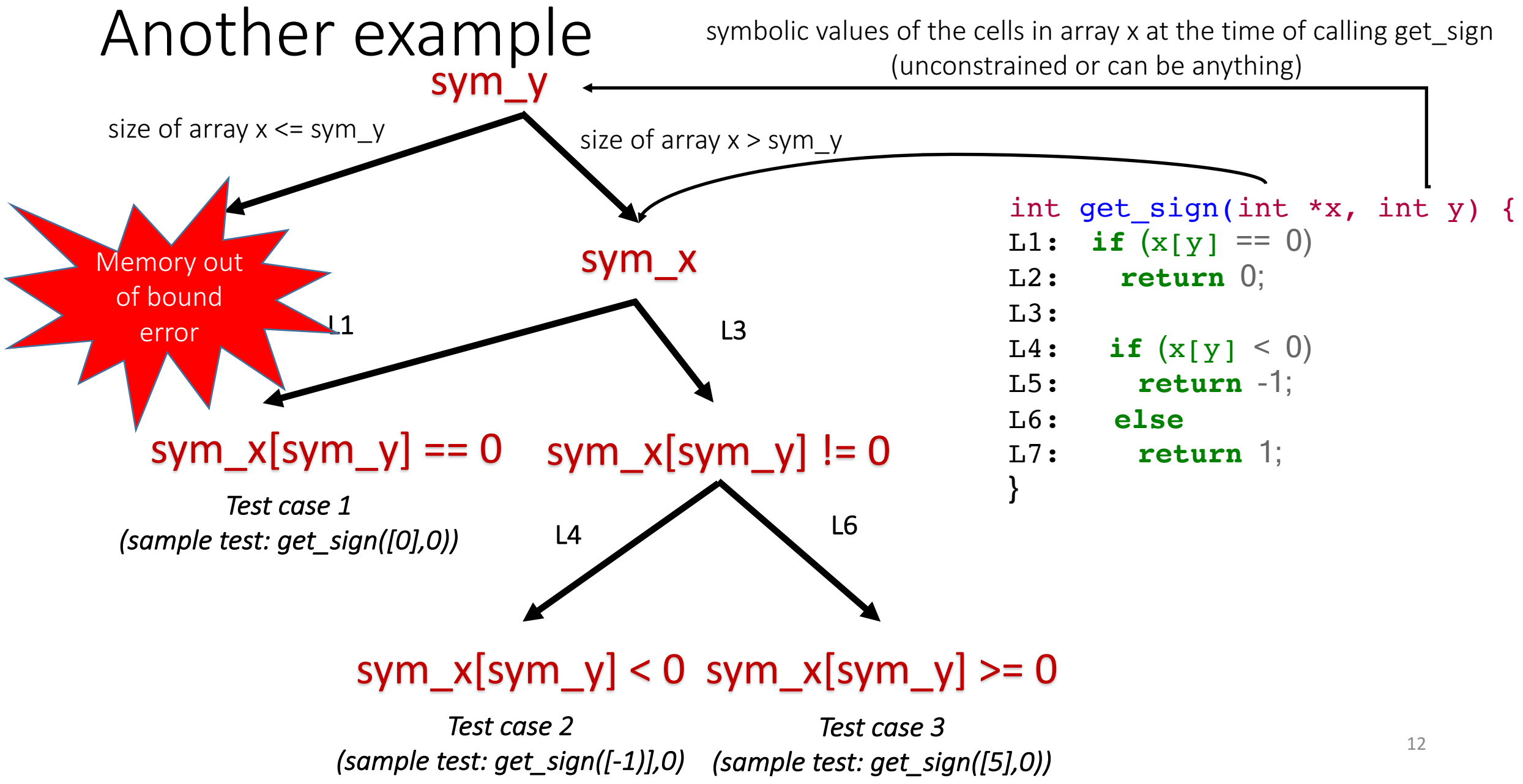
$ KLEE –write-kqueries get_sign.bc
$ ls klee-last/test*
klee-last/test000001.kquery klee-last/test000002.ktest
klee-last/test000001.ktest klee-last/test000003.kquery
klee-last/test000002.kquery klee-last/test000003.ktest

Analyzing get_sign with KLEE

```
$ more klee-last/test000001.kquery
array a[4] : w32 -> w8 = symbolic
(query [(Eq 0
             (ReadLSB w32 0 a))]
       false)

$ more klee-last/test000003.kquery
array a[4] : w32 -> w8 = symbolic
(query [(Eq false
             (Eq 0
                 N0:(ReadLSB w32 0 a)))
         (Slt N0 0)]
        false)


$ more klee-last/test000002.kquery
array a[4] : w32 -> w8 = symbolic
(query [(Eq false
             (Eq 0
                 N0:(ReadLSB w32 0 a)))
         (Eq false (Slt N0 0))]
        false)
```



Another example




```
int get_sign(int *x, int y) {
L1:  if (x[y] == 0)
L2:    return 0;
L3:
L4:   if (x[y] < 0)
L5:     return -1;
L6:   else
L7:     return 1;
}
```

Memory Bugs

- Thanks to the precise memory model used in its dynamic symbolic execution approach, KLEE can detect a variety of memory vulnerabilities/bugs
- NULL pointer dereferencing
- Memory out of bounds reads/writes
- Use-after-free
- Double-free
- Memory leaks

Wrap-up for PART I

- Symbolic execution has emerged as a white-box automated testing technique by quickly became an important technique for
  - Detecting memory and other types of vulnerabilities
  - Reverse engineering and model extraction
- One of the biggest challenges in symbolic execution is the path explosion problem
  - Prevents finding deep bugs
- Another challenge is environment initialization
  - Difficult to perform for complex systems

PART II: PROMPT for API Model Guided Symbolic Execution

Outline for Part II

- Motivation
- PROMPT Overview
- API Modeling with PROSE
- Hands-on Activity: Tutorial Examples

Motivation

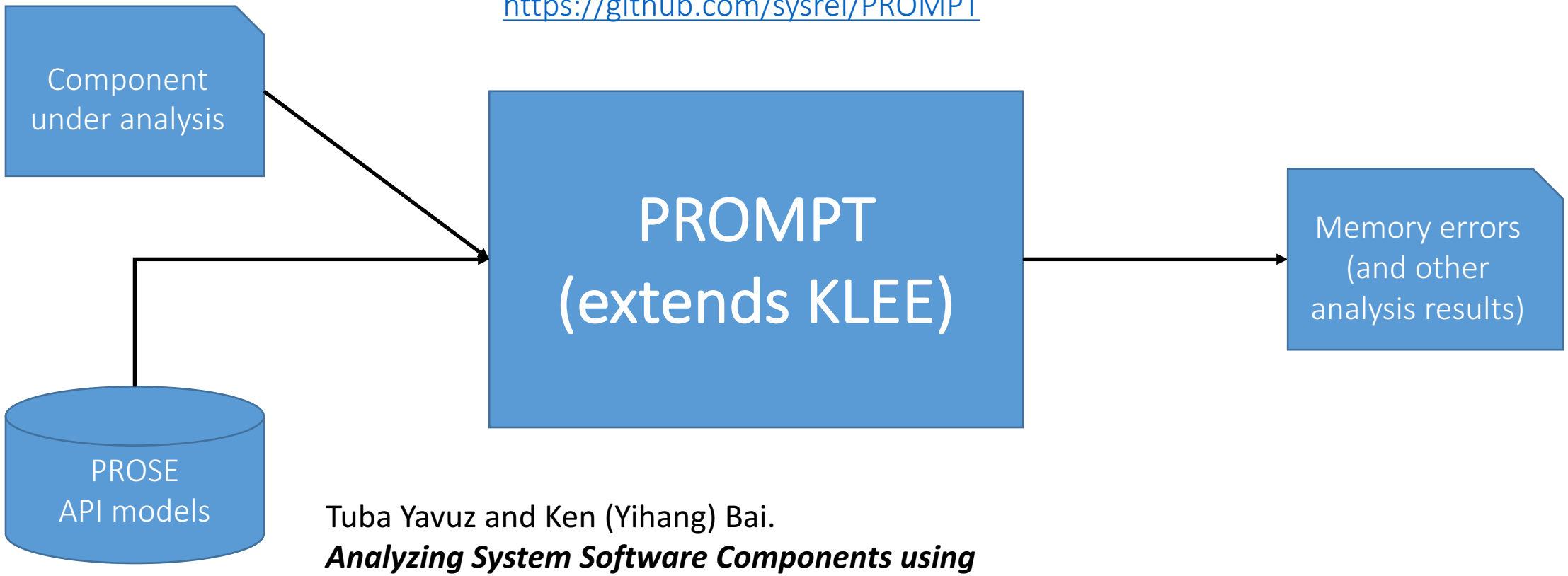
- The modeling of the Application Programming Interface (API) of complex system code is key to the scalable and precise analysis
- Techniques that automatically learn the behavior of APIs are not mature yet
  - Automated synthesis: the state space is often huge, guided by some syntax and/or input/output samples
  - Machine learning: still a big semantic gap between program representations and those used by the learning algorithms

Motivation

- Under-constrained symbolic execution as implemented in UC-KLEE can support component-level analysis
  - David A. Ramos and Dawson Engler. **Under-Constrained Symbolic Execution: Correctness Checking for Real Code. USENIX Security 2015.**
  - **UC-KLEE is not open-source**
  - **The goal of modeling is to filter out false positives**
- **PROMPT approach for component-level analysis for system code**
  - **API modeling and API model guided symbolic execution**
  - Better control for scalability and precision

Solution: PROMPT & PROSE

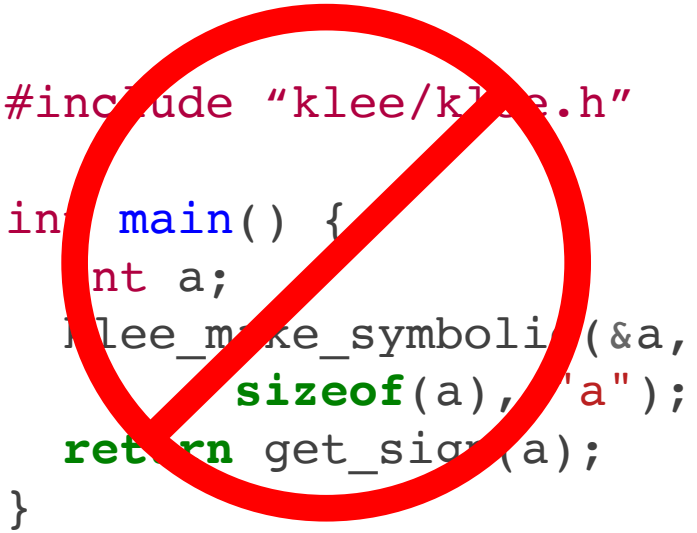
https://github.com/sysrel/PROMPT



Tuba Yavuz and Ken (Yihang) Bai.
Analyzing System Software Components using API Model Guided Symbolic Execution.
To appear in Automated Software Engineering (27:6).
DOI: 10.1007/s10515-020-00276-5

Analyzing get_sign with PROMPT

Test driver



```
#include “klee/klee.h”

int main() {
  int a;
  klee_make_symbolic(&a,
      sizeof(a), "a");
  return get_sign(a);
}
```

The model (model.txt)

```
global settings:
data models:
function models:
lifecycle model:
        entry-point get_sign
```

component under analysis

```
int get_sign(int x) {
L1:  if (x == 0)
L2:    return 0;
L3:
L4:   if (x < 0)
L5:     return -1;
L6:   else
L7:     return 1;
}
```

$ PROMPT –prose-api-model=model.txt –write-kqueries get_sign.bc
$ ls klee-last/test*
klee-last/test000001.kquery klee-last/test000002.ktest
klee-last/test000001.ktest   klee-last/test000003.kquery
klee-last/test000002.kquery klee-last/test000003.ktest

Analyzing get_sign with PROMPT

```
$ more klee-last/test000001.kquery
array get_sign_0_arg_0[4] : w32 -> w8 = symbolic
(query [(Eq 0
           (ReadLSB w32 0 get_sign_0_arg_0))]
       false)
```

```
$ more klee-last/test000003.kquery
array get_sign_0_arg_0[4] : w32 -> w8 = symbolic
(query [(Eq false
             (Eq 0
                 N0:(ReadLSB w32 0 get_sign_0_arg_0)))
         (Slt N0 0)]
        false)
```

```
$ more klee-last/test000002.kquery
array get_sign_0_arg_0[4] : w32 -> w8 = symbolic
(query [(Eq false
             (Eq 0
                 N0:(ReadLSB w32 0 get_sign_0_arg_0)))
         (Eq false (Slt N0 0))]
        false)
```

component under analysis

```
int get_sign(int x) {
L1:  if (x == 0)
L2:    return 0;
L3:
L4:   if (x < 0)
L5:     return -1;
L6:   else
L7:     return 1;
}
```

PROMPT: API Model Guided Symbolic Execution

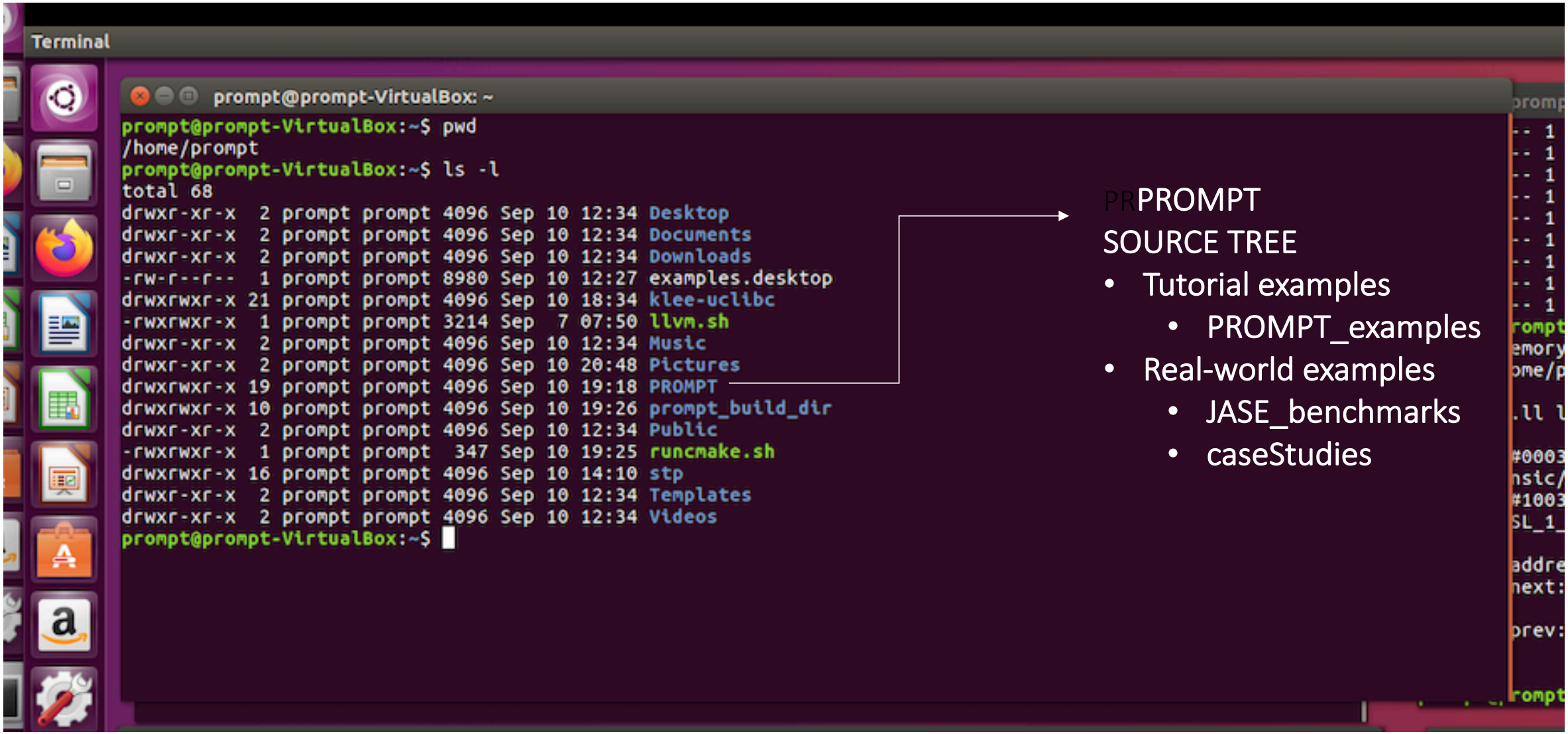
- Performs under-constrained symbolic execution
  - No need for a test driver
- Performs lazily initialization according to the specified API model
- Extends the KLEE symbolic execution engine to incorporate API modeling rules during handling of various instructions
  - Memory access instructions
  - Function calls
  - Return instructions
- Has been applied to real-world system code
  - Linux device drivers, cryptographic libraries, BlueZ

API Modeling with PROSE

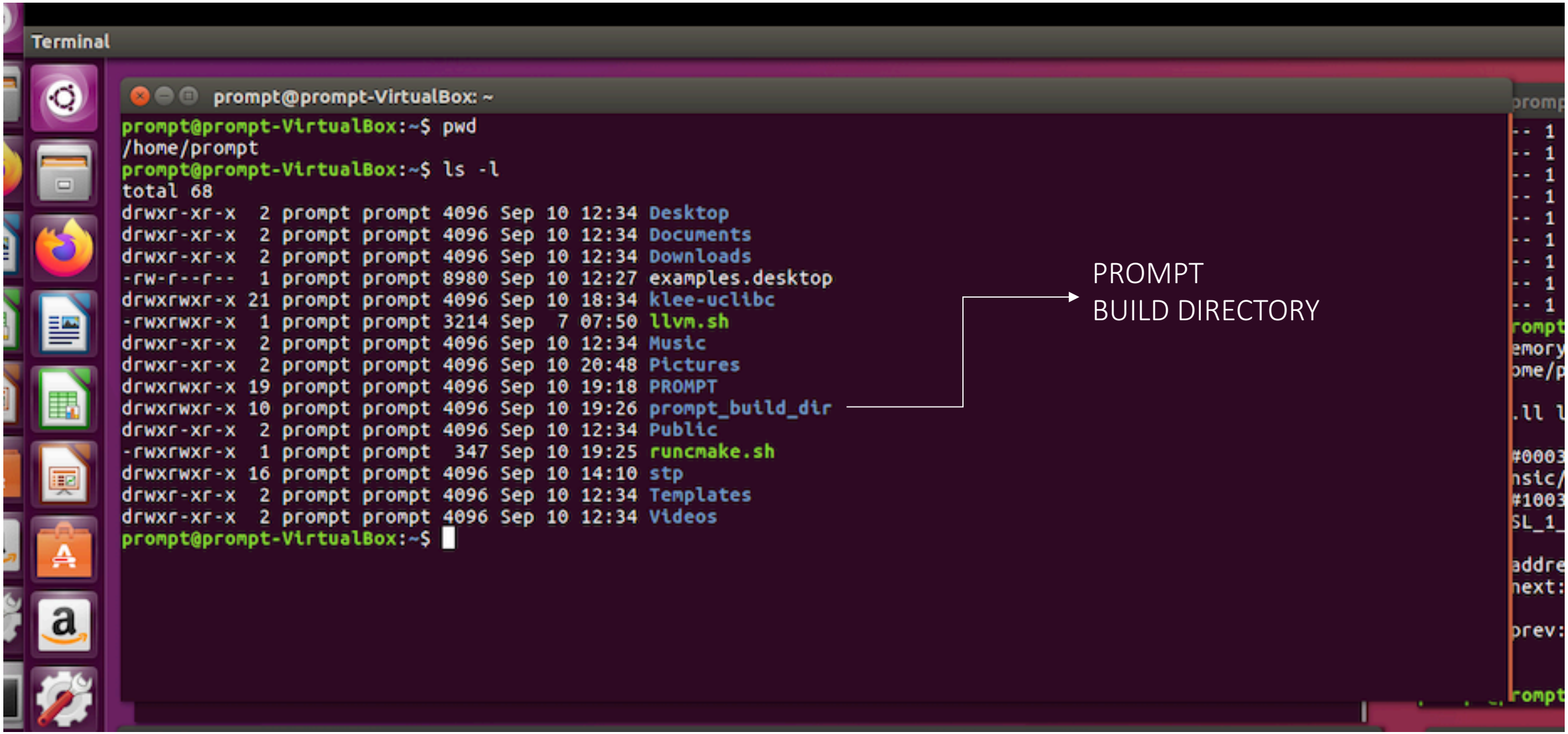
- The manual for PROSE language can be found at
  - https://github.com/sysrel/PROMPT/blob/master/docs/PROMPT_Manual.txt
- A template PROSE model can be found at
  - https://github.com/sysrel/PROMPT/blob/master/docs/PROSE_template.txt
- Four types of specifications
  - Global settings
  - Data modeling
  - Function modeling
  - Life-cycle rules
- Tutorial examples
  - https://github.com/sysrel/PROMPT/tree/master/PROMPT_examples

Hands-on Activity: Tutorial Examples

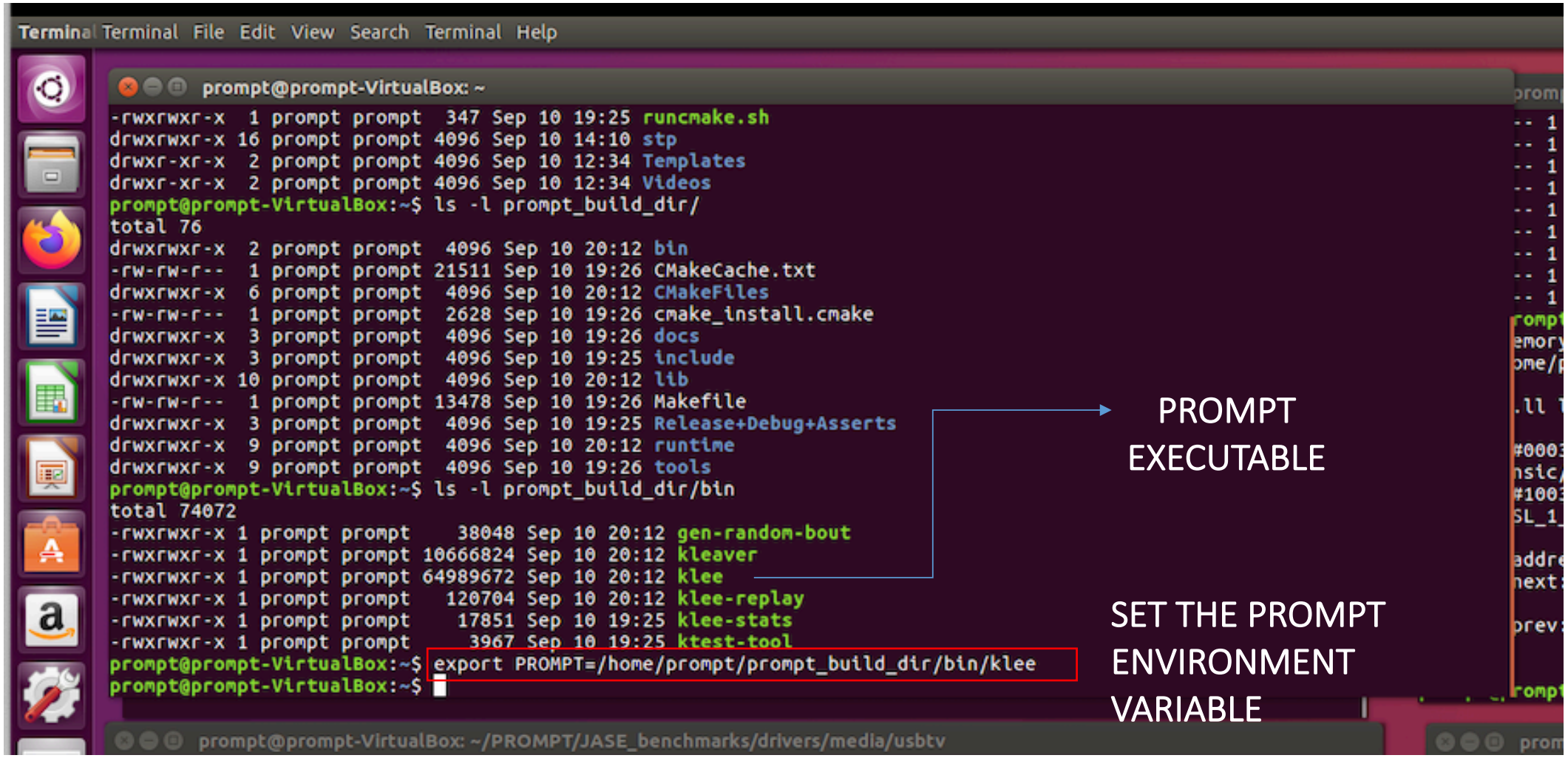
Using the virtual machine image



Using the virtual machine image



Getting ready for running the examples



Format for our examples

PROSE API MODEL

```
global settings: …
data models: …
function models: …
lifecycle model:
       entry-point …
```

COMPONENT UNDER ANALYSIS

SOME C CODE

STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd PATH_TO_THE_EXAMPLE
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: ?
What to expect (error/no error) on certain line numbers
All test cases will be written to some test* file under the klee-last directory
Those that end with the .err suffix are the error paths

Modeling with PROSE

- **Global settings**
- Data modeling
- Function modeling
- Lifecycle modeling

Specifying the Size of a Pointer Argument

PROSE API MODEL

```
global settings:
       array size 5;
data models:
function models:
lifecycle model:
       entry-point foo
```

COMPONENT UNDER ANALYSIS

```
void foo(int *a) {
L1:       a[4] = 11;
L2:       a[5] = 10;
}
```

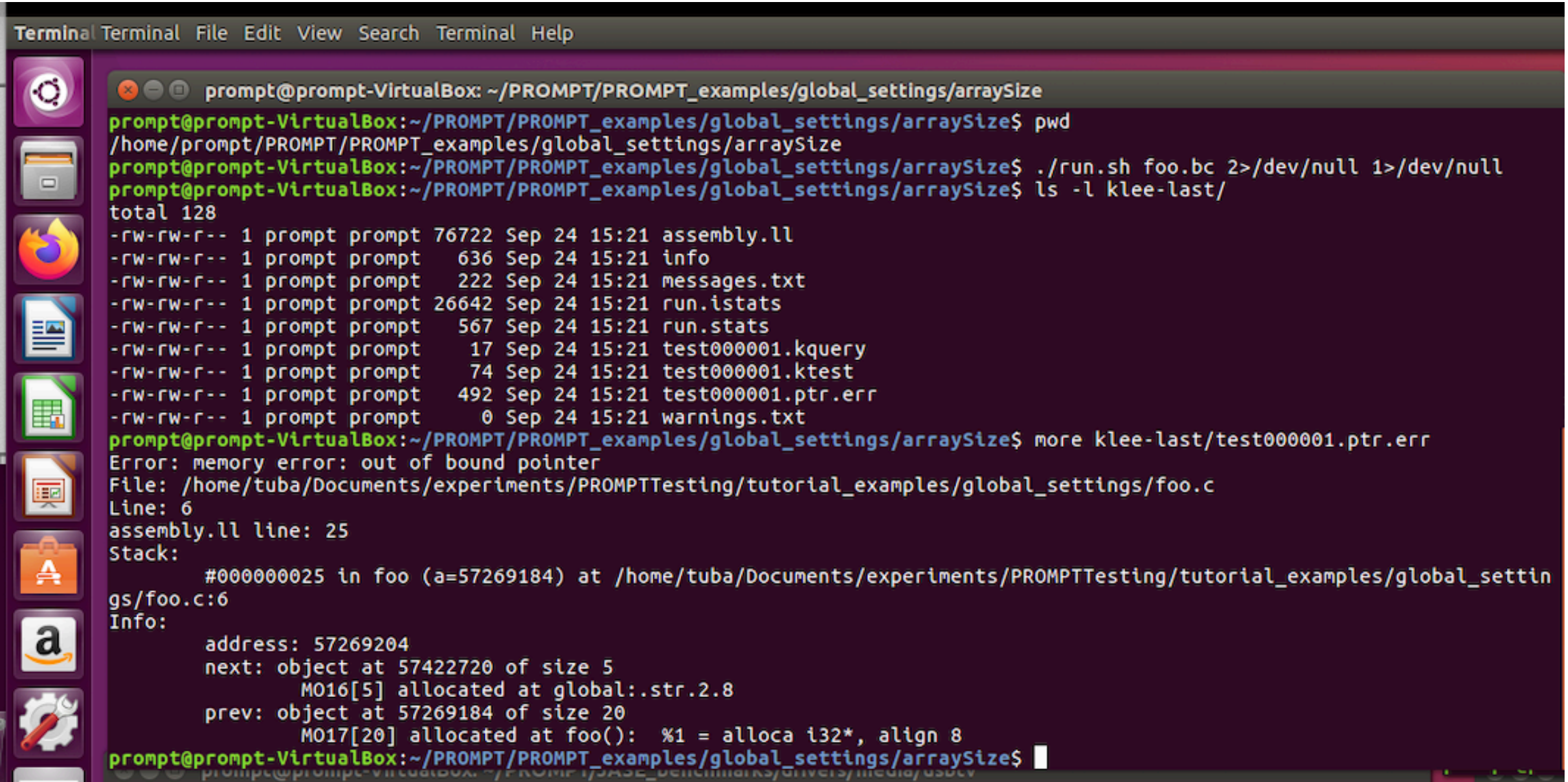
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd global_settings/arraySize/
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

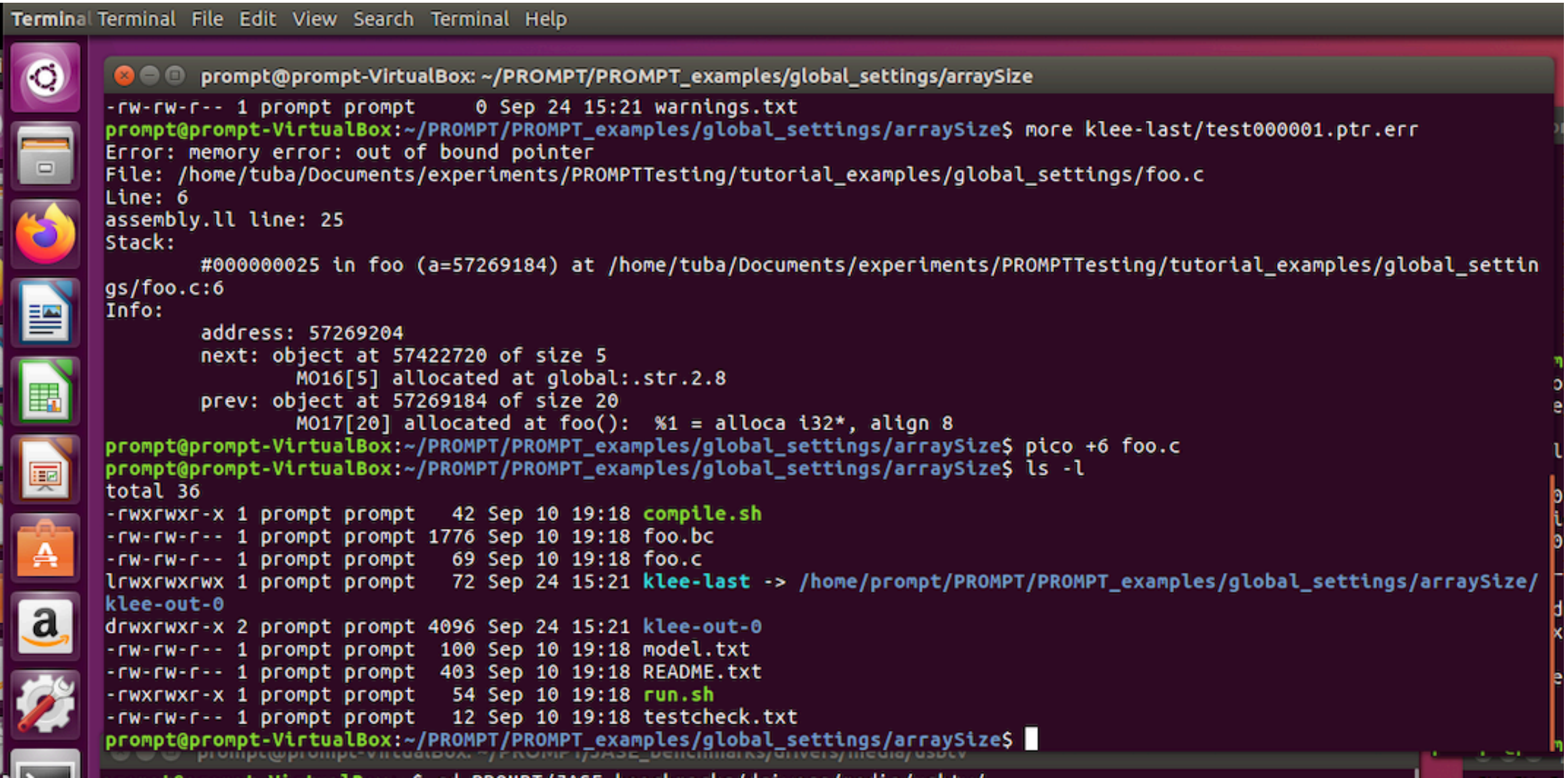
PROMPT RESULTS:

# of paths: 1
No memory error at line L1
Memory error at line L2

Let’s try it on the VM!



Let’s try it on the VM!



Specifying initialization of function pointers to NULL

PROSE API MODEL

```
global settings:
       init funcptrs to null on;
data models:
function models:
lifecycle model:
       entry-point foo
```

COMPONENT UNDER ANALYSIS

```
struct A {
        int (*bar)();
        int a;
       };
```

```
typedef struct A A_t;

void foo(A_t *a1) {
 L1: if (a1->bar)
 L2:    assert(0);
 L3: int b = a1->a;
}
```

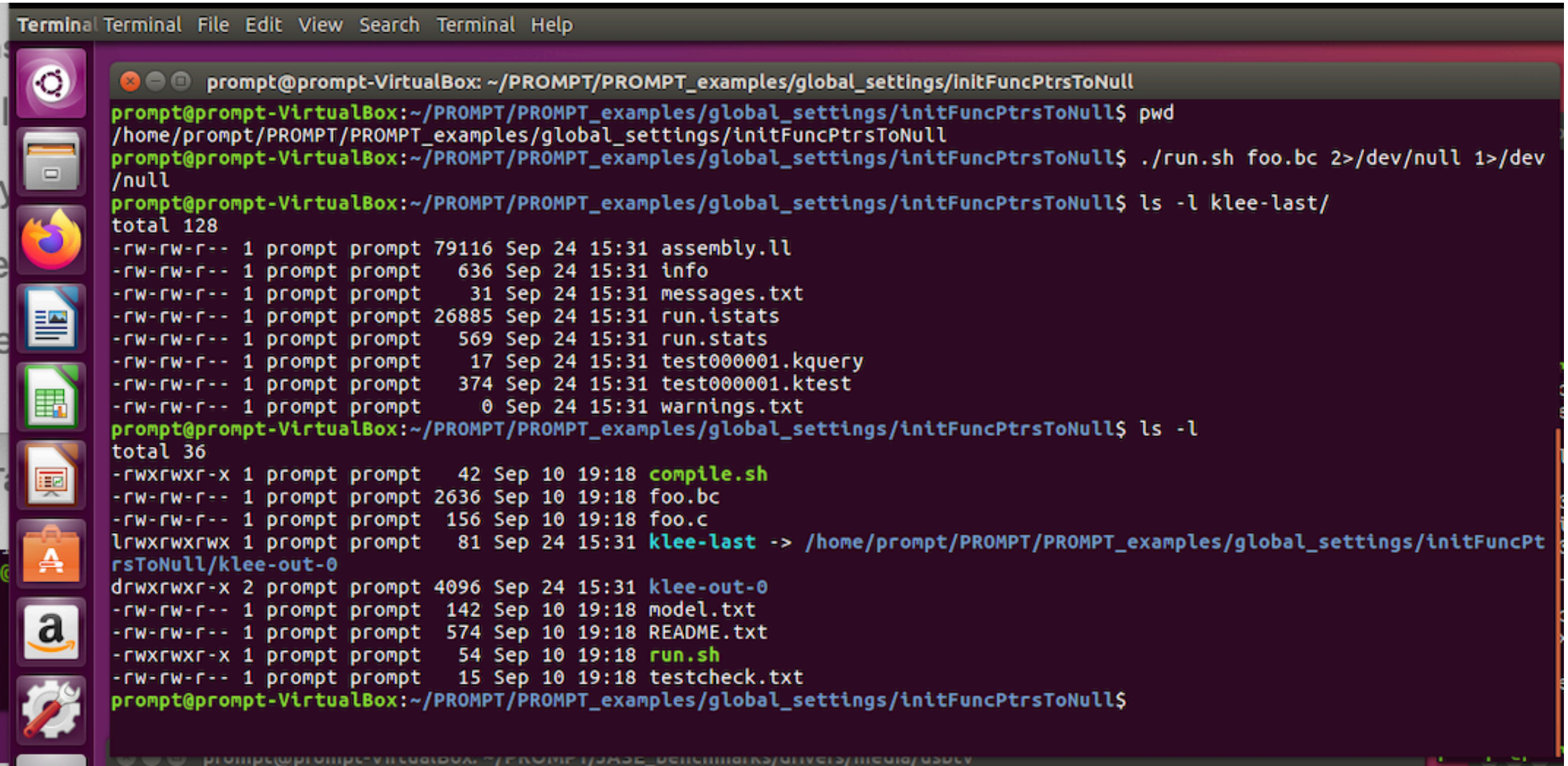
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd global_settings/initFuncPtrsToNull
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 1
No assertion error at line L2

Let’s try it on the VM!



Specifying modeling of functions with inline assembly

PROSE API MODEL

```
global settings:
        model funcs with asm on
but_use_original_with_pattern bar
                  except bar1;
        symbolize inline asm on;
data models:
function models:
lifecycle model:
        entry-point foo
```

COMPONENT UNDER ANALYSIS

```
int foo1() {__asm__(…);return 0;}

int bar() {__asm__(…);return 0;}

int bar1() {__asm__(…);return 1;}
```

```
void foo(int *a) {
L1:         int y = foo1();
L2:         if (y)
L3:           assert(0);
L4:         int z = bar();
L5:         if (z)
L6:           assert(0);
L7:         int w = bar1();
L8:         if (w != 1)
L9:           assert(0); }
```

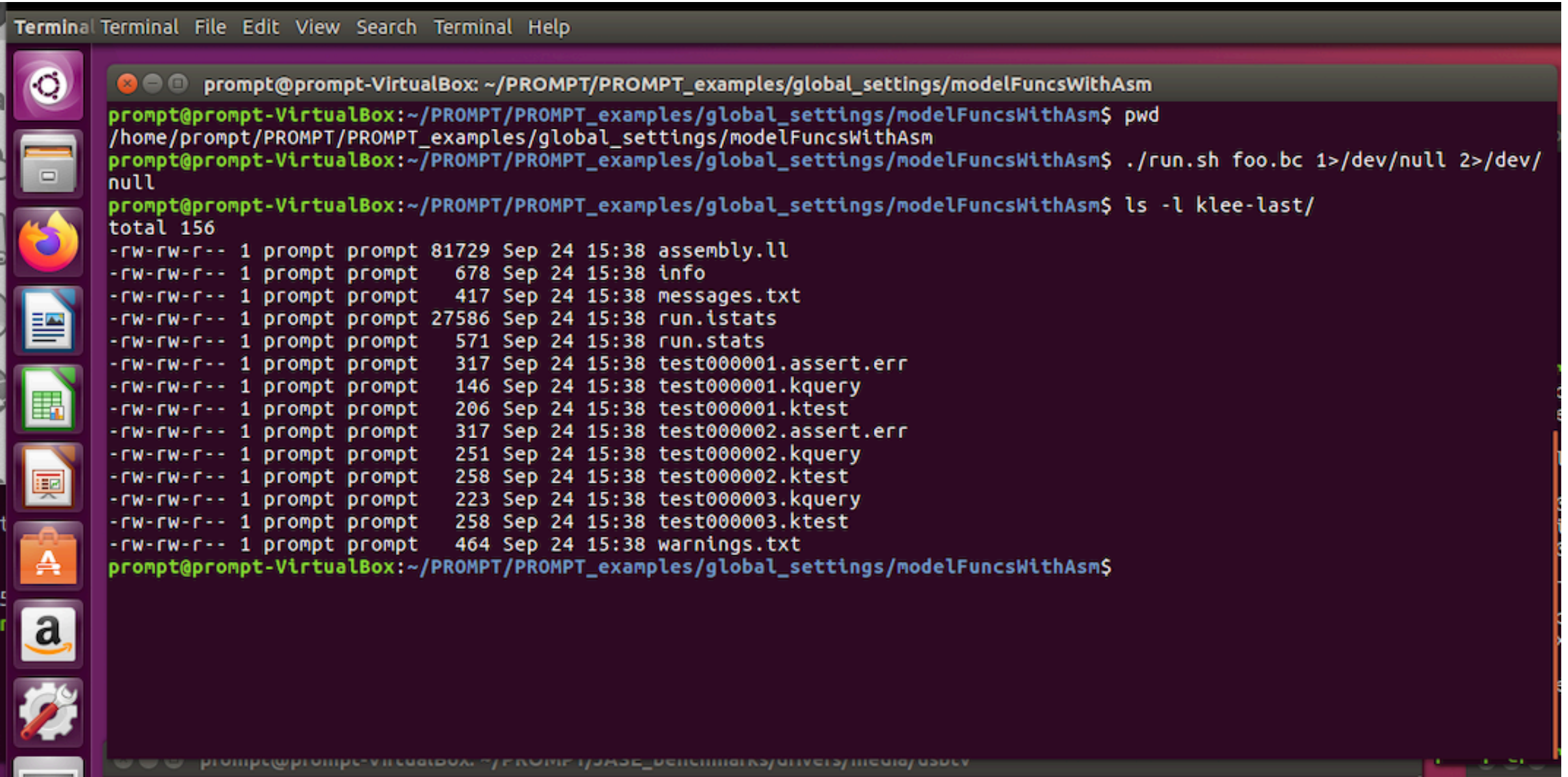
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd global_settings/modelFuncsWithAsm
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

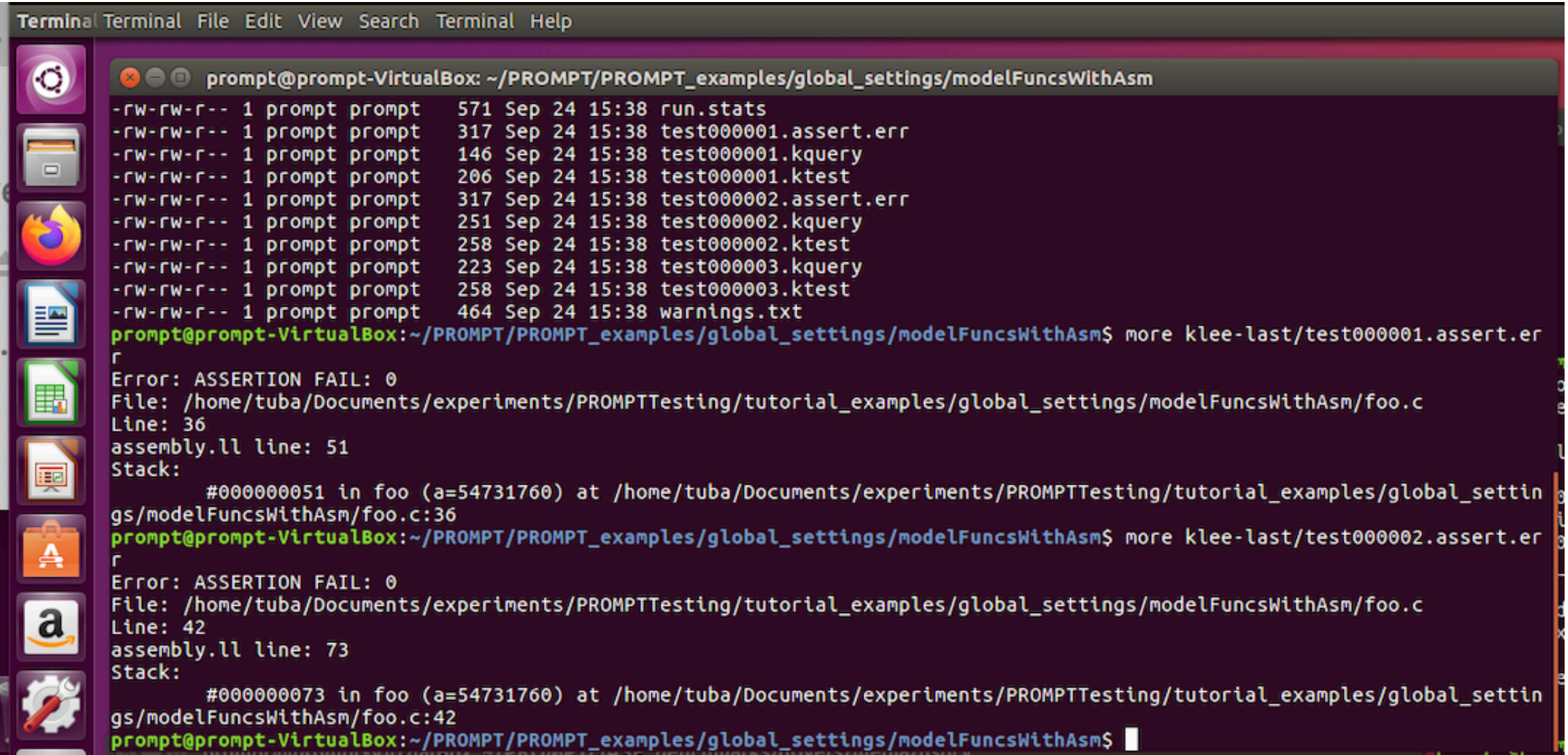
PROMPT RESULTS:

of paths: 3
Assertion error at line L3
No assertion error at line L6
Assertion error at line L9

Let’s try it on the VM!



Let’s try it on the VM!



Specifying NULL return option for modeled functions

PROSE API MODEL

```
global settings:
       null return on;
data models:
function models:
lifecycle model:
       entry-point foo
```

COMPONENT UNDER ANALYSIS

```
int *bar();

void foo(int *a) {
L1:  int *y = bar();
L2:  int z = *y;
}
```

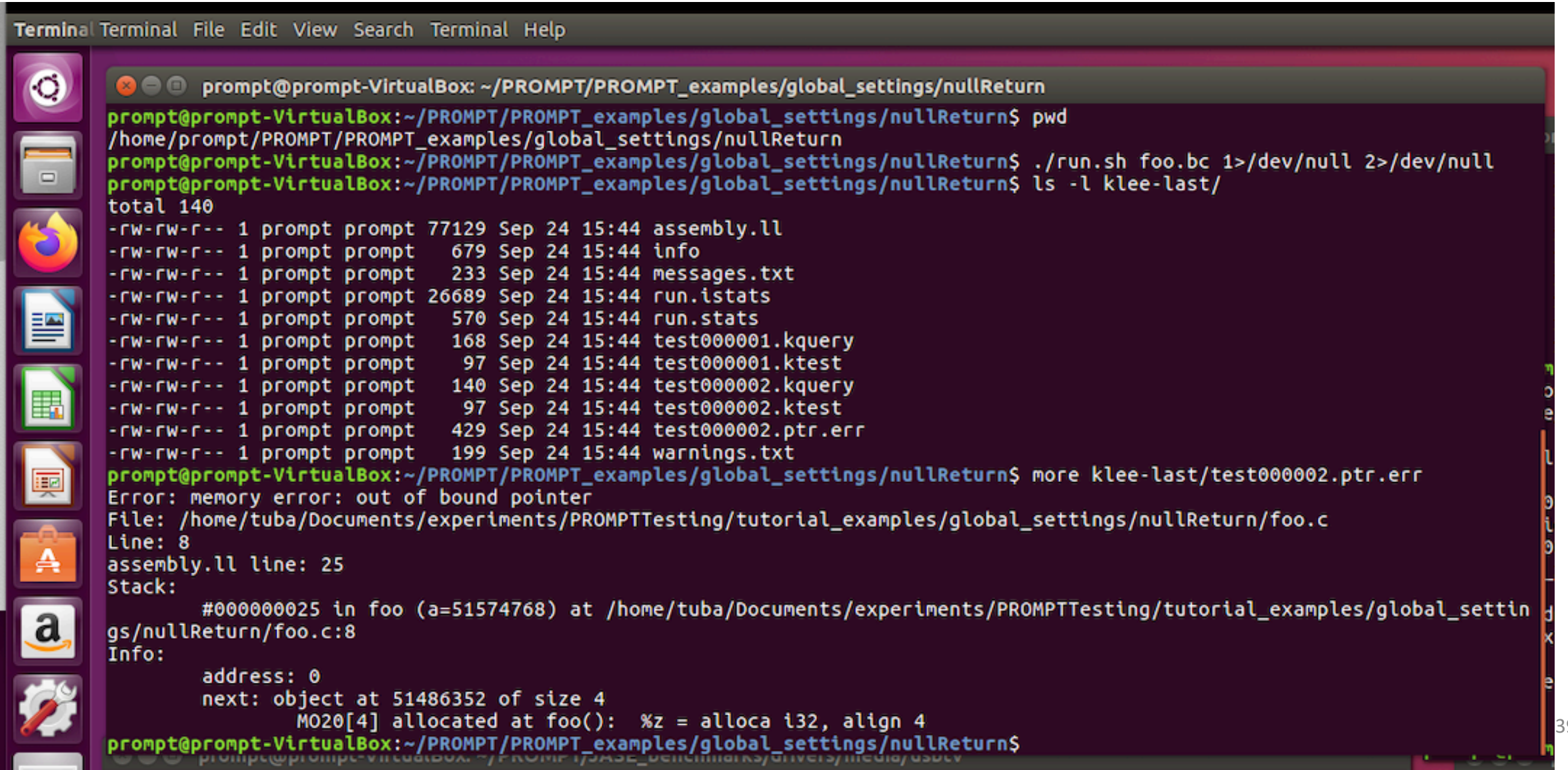
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd global_settings/nullReturn
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: 2
Memory error at L2

Let’s try it on the VM!



Specifying skipping of havocing singleton argument types

PROSE API MODEL

```
global settings:
        skip havocing singletons on;
data models:
        singleton A;
function models:
lifecycle model:
        entry-point foo
```

COMPONENT UNDER ANALYSIS

```
struct A {
  int a;
};
struct B {
  int b;
};
void bar(struct A *,
struct B *);
```

```
void foo(struct A *a1,
         struct B *b1) {
  L1:  a1->a = 5;
  L2:  b1->b = 10;
  L3:  bar(a1, b1);
  L4:  if (a1->a != 5)
  L5:    assert(0);
  L6:  if (b1->b != 10)
  L7:    assert(0);}
```

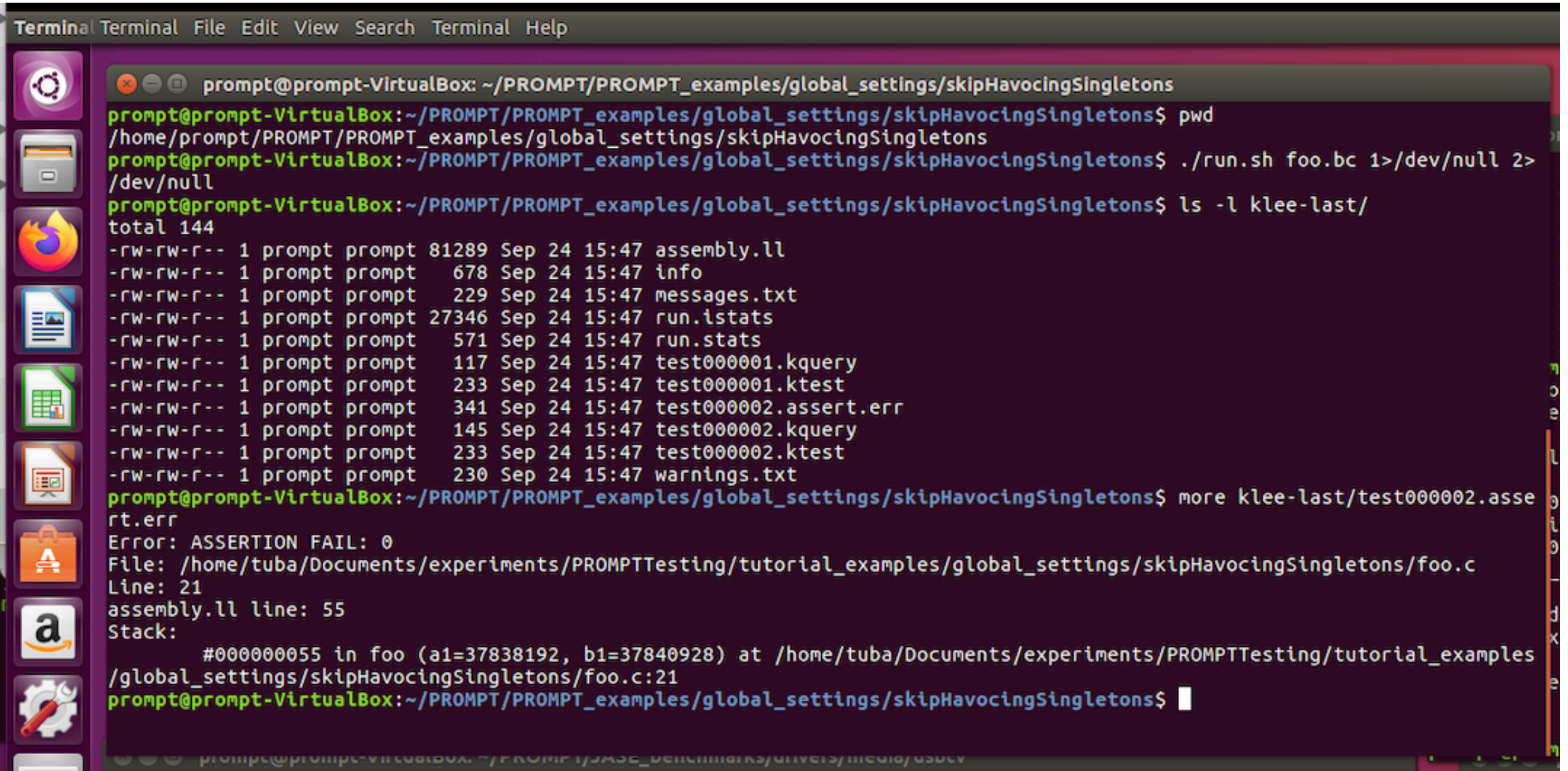
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd global_settings/skipHavocingSingletons
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 2
No assertion failure at line L5
Assertion failure at line L7

Let’s try it on the VM!



Modeling with PROSE

- Global settings
- **Data modeling**
- Function modeling
- Lifecycle modeling

Specifying initialization of a function pointer field with some function

PROSE API MODEL

```
global settings:
data models:
  (bar = x) where bar is function,
                  x is B field 2;
  (x > 0) where x is foo arg 0;
function models:
lifecycle model:
       entry-point foo
```

COMPONENT UNDER ANALYSIS

```
struct B {
  int b1;
  int b2;
  void (*f)(void);
};
void bar()
 {L4: assert(0);}
```

```
void foo(int a, struct B *b) {
L1:      if (a < 0)
L2:         assert(0);
L3:      b->f();
}
```

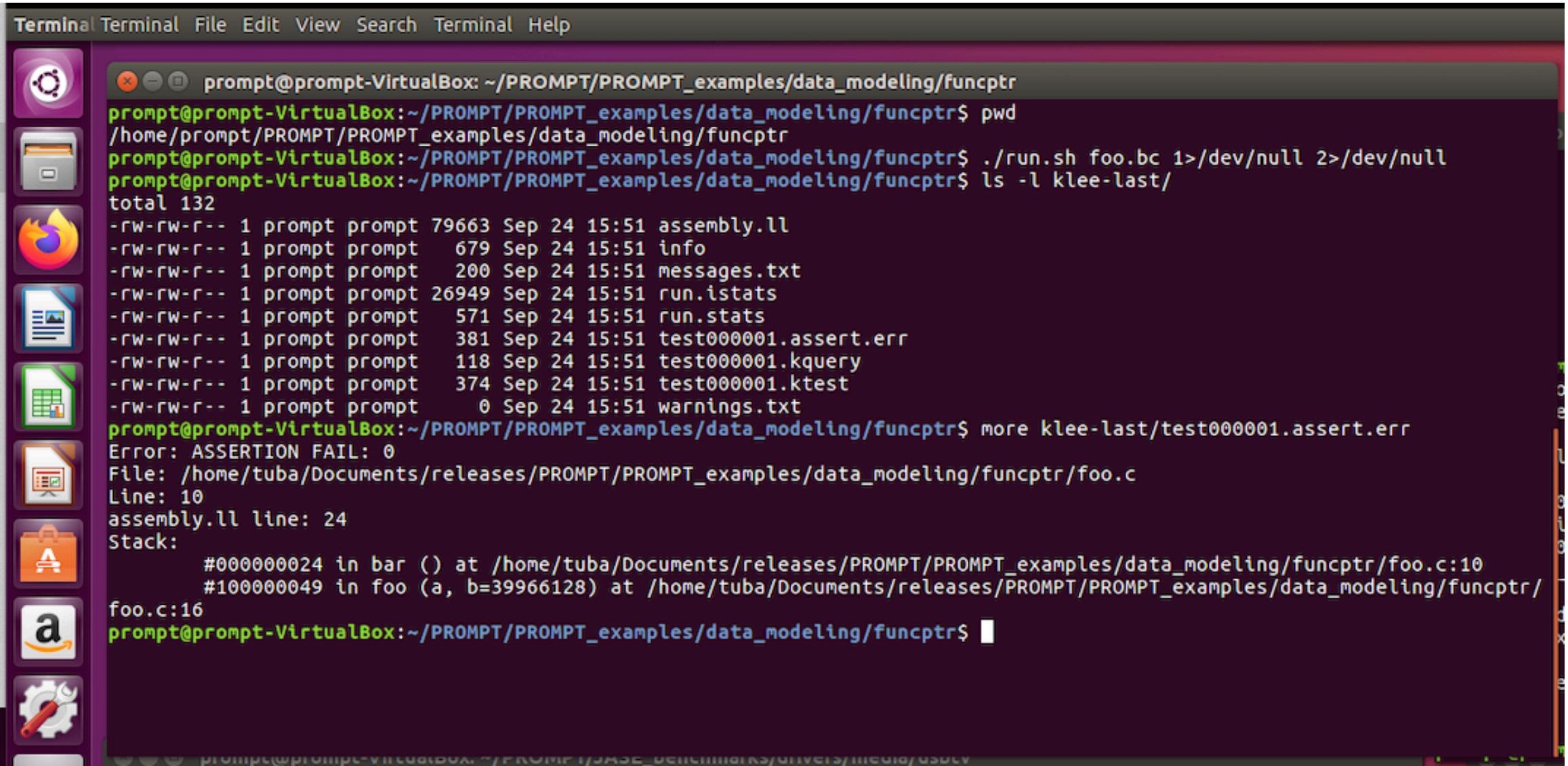
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/funcptr
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 1
No assertion failure at line L2
No invalid function pointer at L3
Assertion failure at line L4

Let’s try it on the VM!



Specifying type embedding relationships

PROSE API MODEL

```
global settings:
data models:
        type A embeds type B;
function models:
lifecycle model:
        entry-point foo
```

COMPONENT UNDER ANALYSIS

```
#define cont_of(ptr, type, member) ({ // container_of
    void *__mptr = (void *)(ptr);
    ((type *)(__mptr - offsetof(type, member)));})

struct A {
  int a;
  struct B b;
  char c;
};

int foo(int x, struct B *b) {struct A *ep;
   L1: ep = cont_of(b,struct A,b);
   L2: if (x > 0)
   L3:     ep->a = x;
   L4: else
   L5:     ep->a = -x;
   L6: return ep->a;
}
```

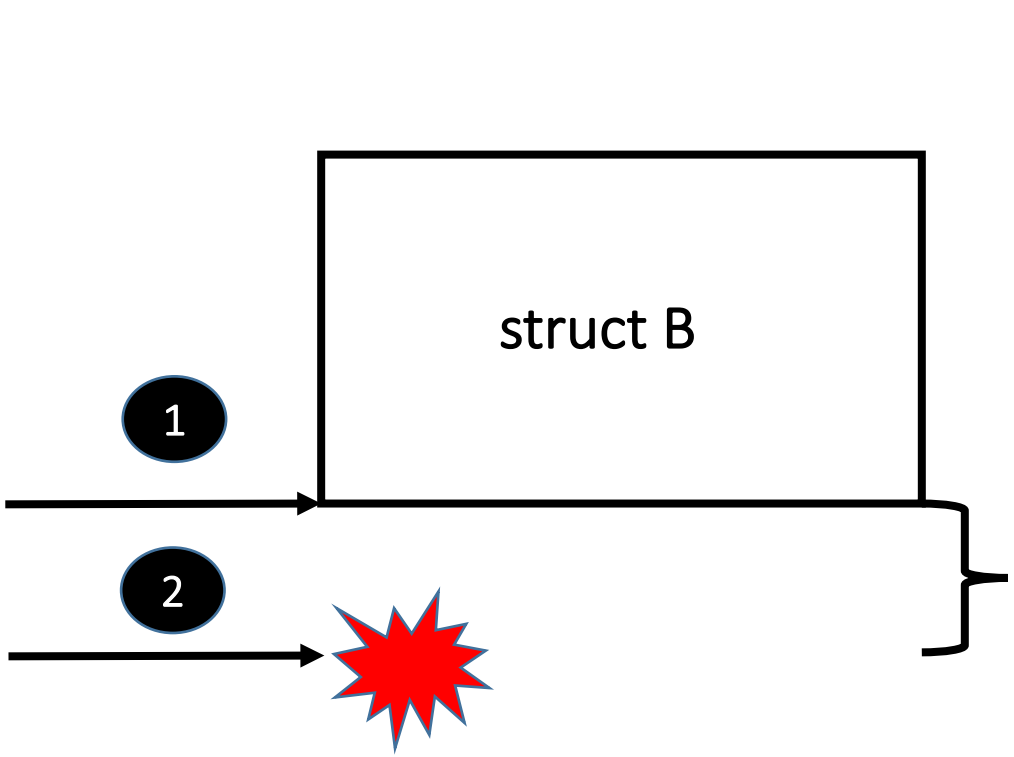
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/embeds
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

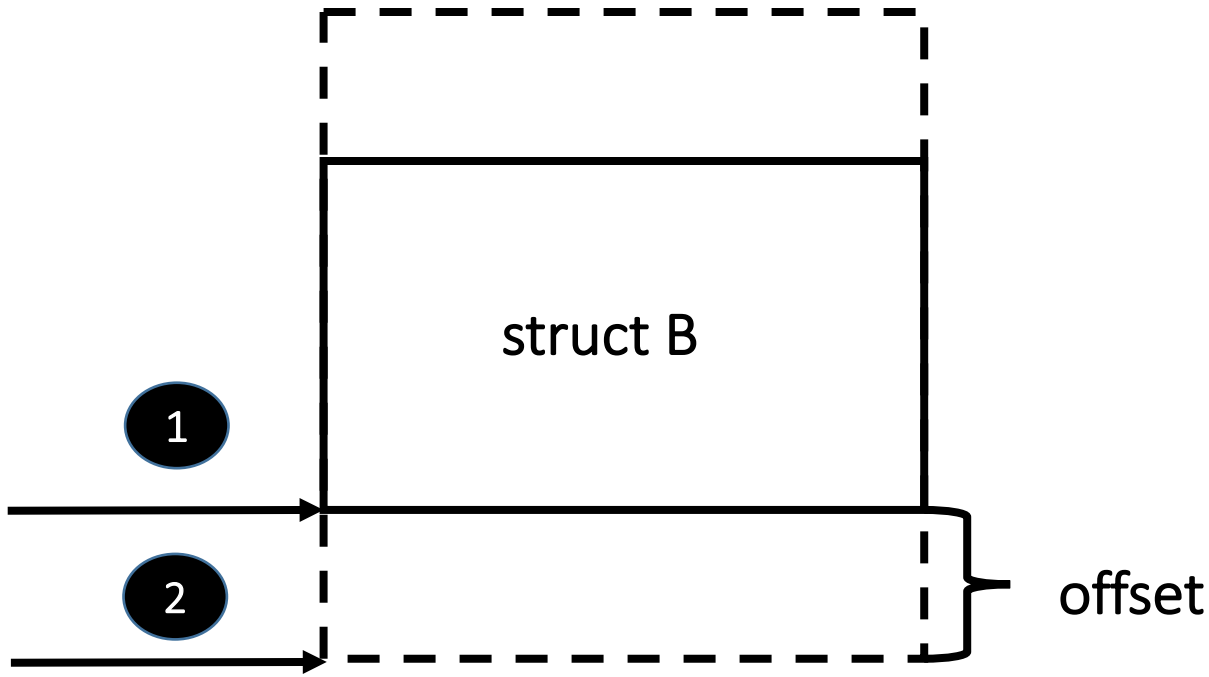
of paths: 2
No memory errors at lines L3, L5, and L6

Pointer arithmetic using a negative offset



Memory out of bounds error when dereferenced!

(False positive due to imprecise context!)



2 = 1 - offset

Just tell PROMPT that type A embeds type B

Specifying type embedding relationships

PROSE API MODEL

```
global settings:
data models:
        type A embeds type B;
function models:
lifecycle model:
        entry-point foo
```

COMPONENT UNDER ANALYSIS

```
#define cont_of(ptr, type, member) ({ // container_of
    void *__mptr = (void *)(ptr);
    ((type *)(__mptr - offsetof(type, member)));})
```

```
struct A {
  int a;
  struct B b;
  char c;
};
```

```
int foo(int x, struct B *b) {
   struct A *ep;
   L1: ep = cont_of(b,struct A,b);
   L2: if (x > 0)
   L3:     ep->a = x;
   L4: else
   L5:     ep->a = -x;
   L6: return ep->a;       }
```

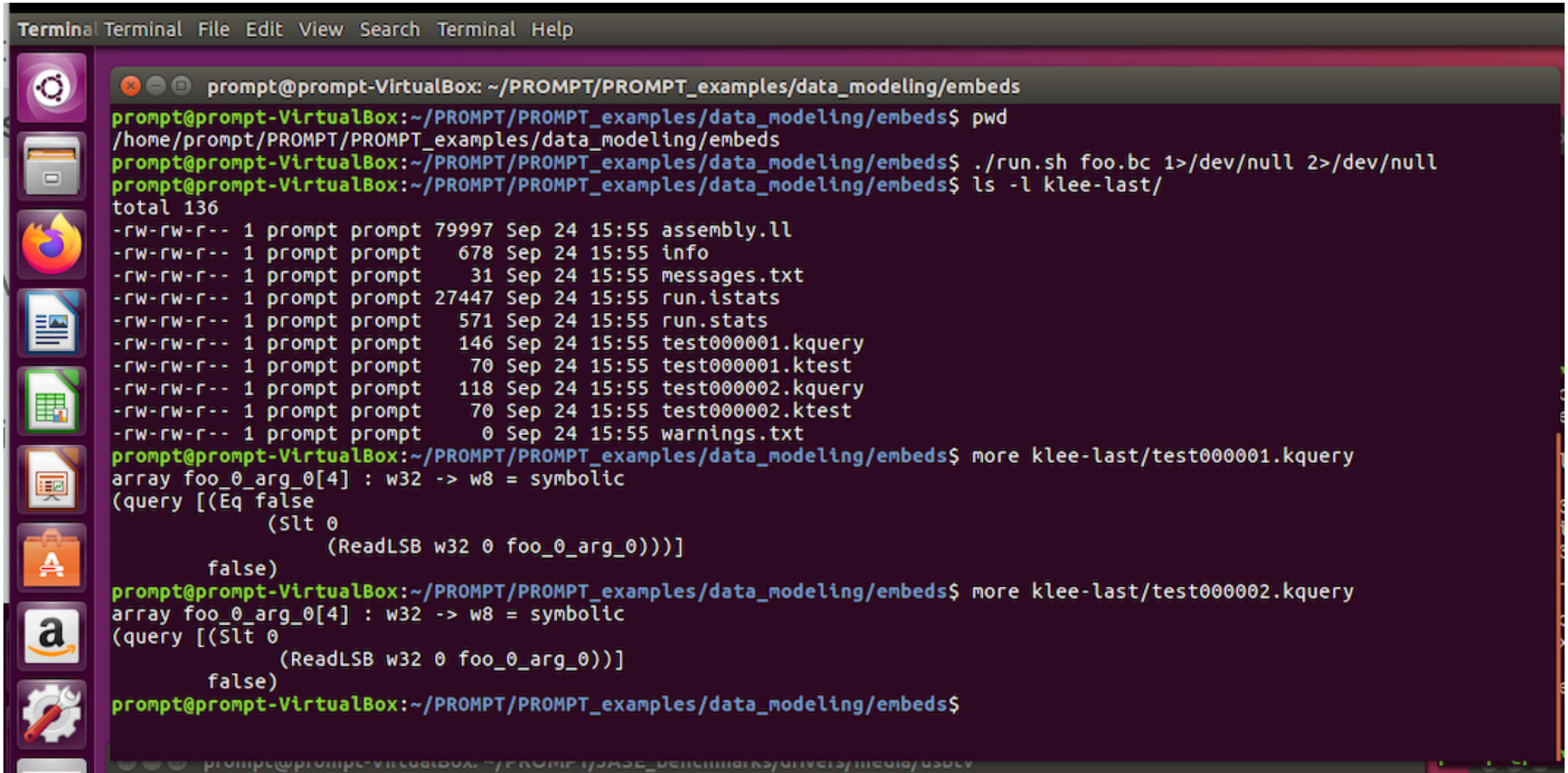
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/embeds
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 2
No memory errors at lines L3, L5, and L6

Let’s try it on the VM!



Specifying a constant size for a function argument of a pointer type

PROSE API MODEL

```
global settings:
data models:
  (35 = w) where w is
                argsize foo arg 1;
function models:
lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
void foo(int a, int *a2) {
L1:       a2[34]=4;
}
```

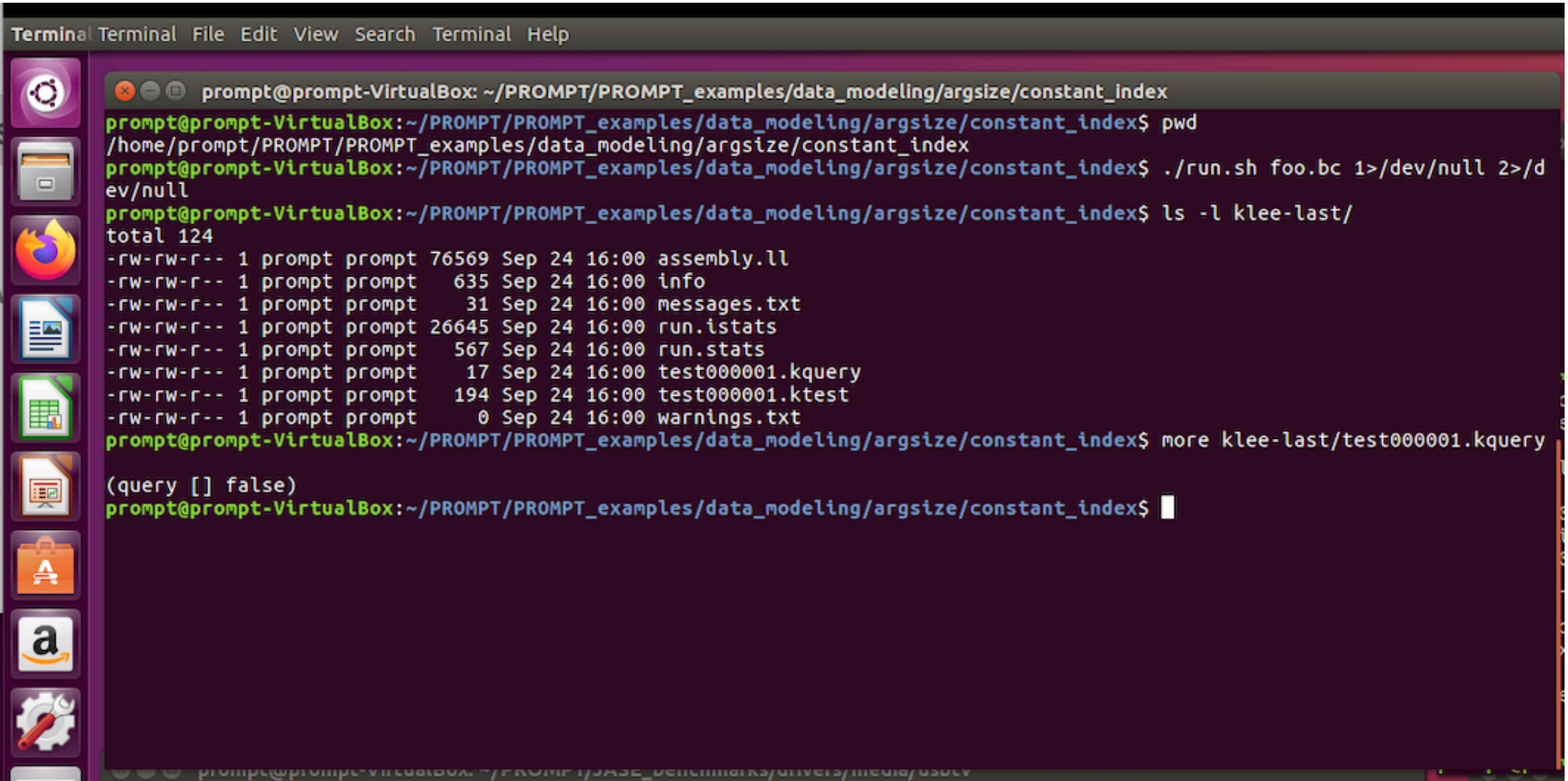
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/arg_size/constant_index
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: 1
No memory error a line L1

Let’s try it on the VM!



Specifying the size for a function argument of a pointer type with a simple equality

PROSE API MODEL

```
global settings:
data models:
  (x = w) where x is foo arg 0,
          w is argsize foo arg 1;
function models:
lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
void foo(int a, int *a2) {
L1:      a2[a-1]=4;
}
```

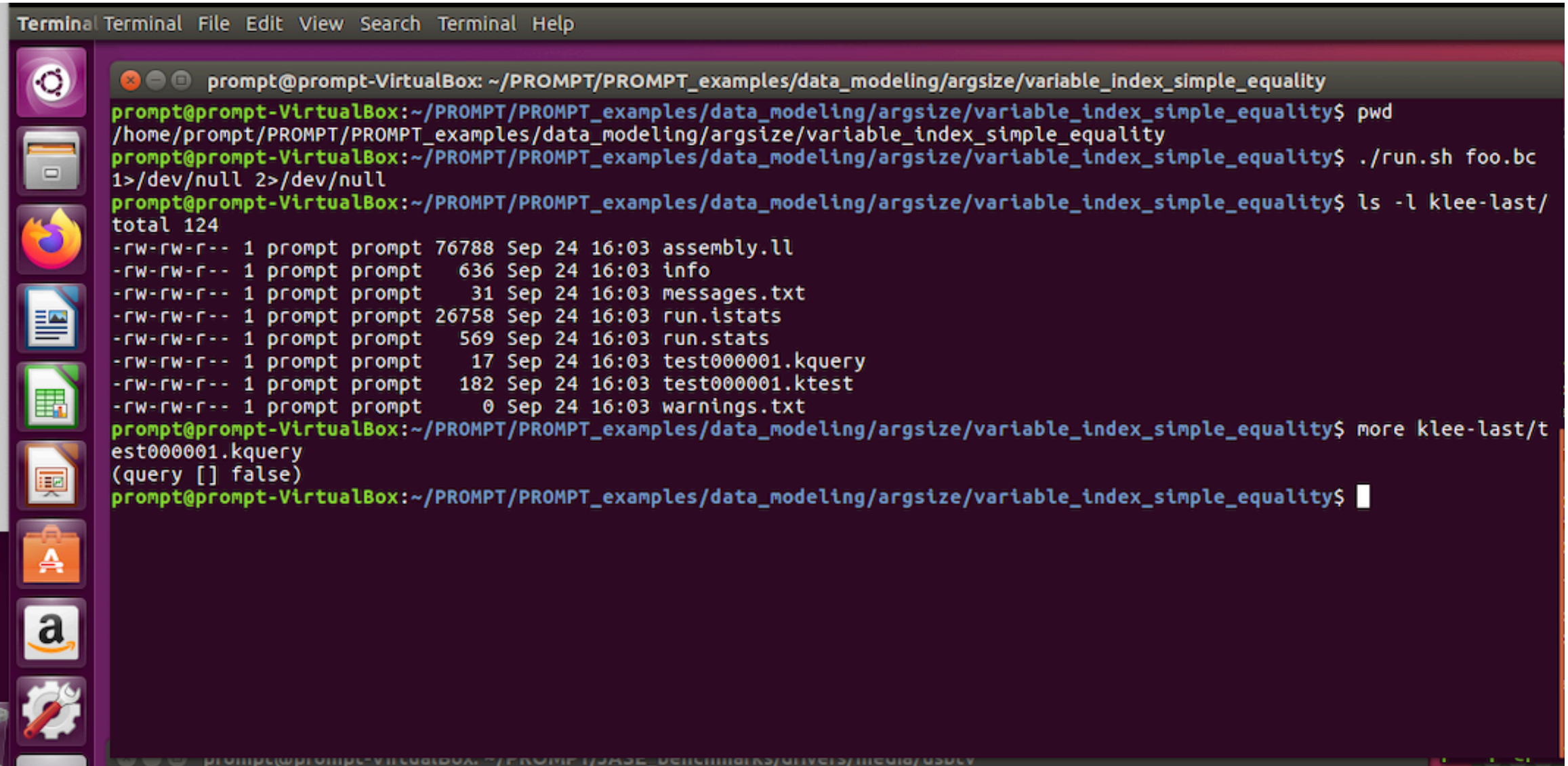
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/arg_size/variable_index_simple_equality
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: 1
No memory error at line L1

Let’s try it on the VM!



Specifying a variable size for a function argument of a pointer type

PROSE API MODEL

```
global settings:
data models:
  ((x + y) < w) where x is foo arg 0,
                      y is foo arg 1,
                  w is argsize foo arg 2;
  ((x + y) > 0) where x is foo arg 0,
                      y is foo arg 1;
function models: lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
void foo(int a, int b, int *a2) {
L1:       a2[a+b-1]=4;
}
```

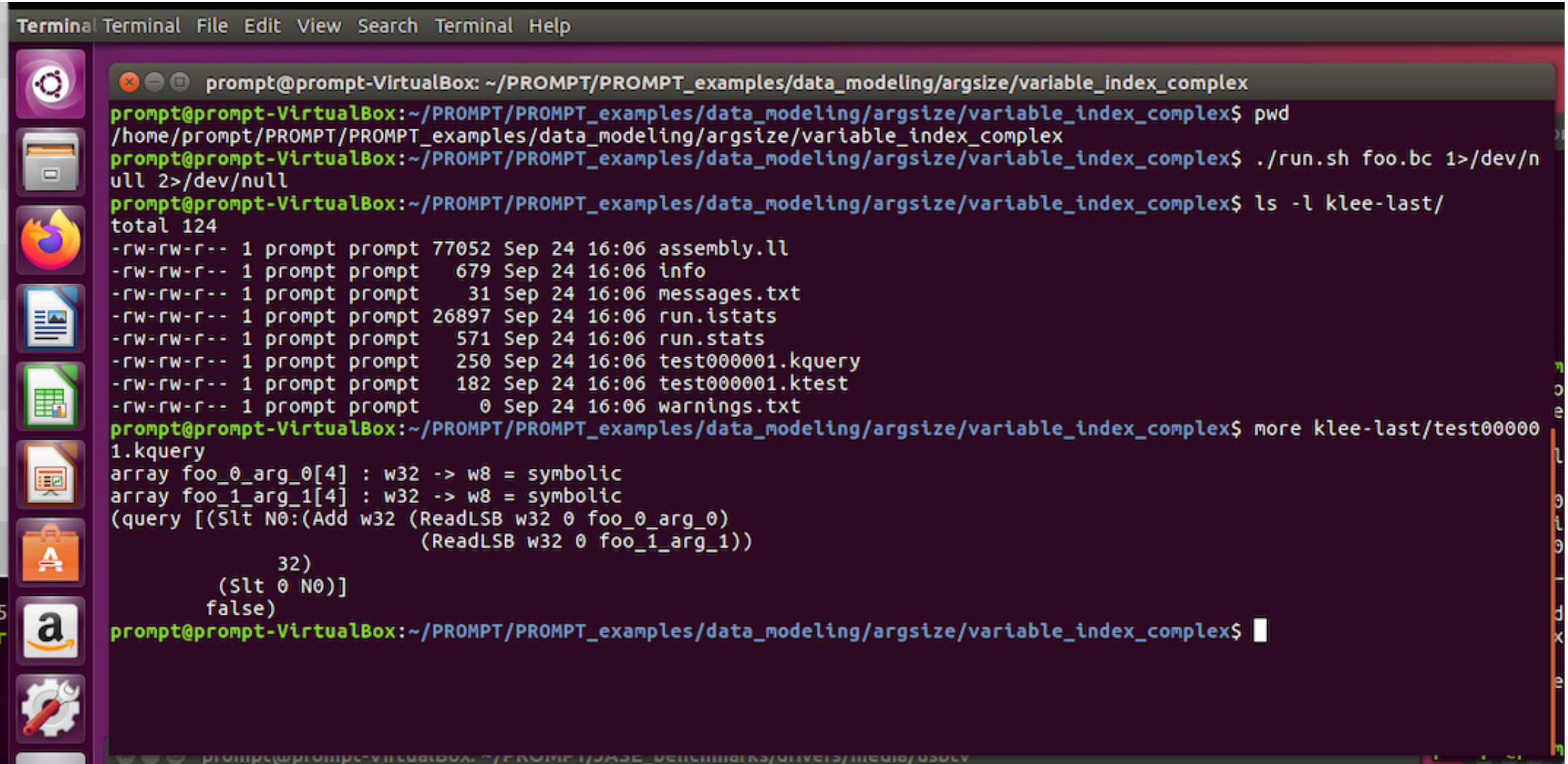
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/arg_size/variable_index_complex
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 1
No memory error at line L1

Let’s try it on the VM!



Specifying data constraints on the struct fields

PROSE API MODEL

```
global settings:
data models:

   (x > 24) where x is foo arg 0;

   (x = 24) where x is foo arg 1;

   (25 = w) where w is sizeof A field 2;

function models:
lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
struct B {
  int b1;
  int b2;
  void (*f)(void);
};

struct A {
  int a; // used as length
  int c;
  struct B *b; // field 2
};
```

```
void foo(int a, int b, struct A
*a2) {
      L1:  if (a <= 24)
      L2:     assert(0);
      L3:  if (b != 24)
      L4:     assert(0);
      L5:  a2->b[b].b1=4;
      L6: a2->b[a].b1=7;
      }
```

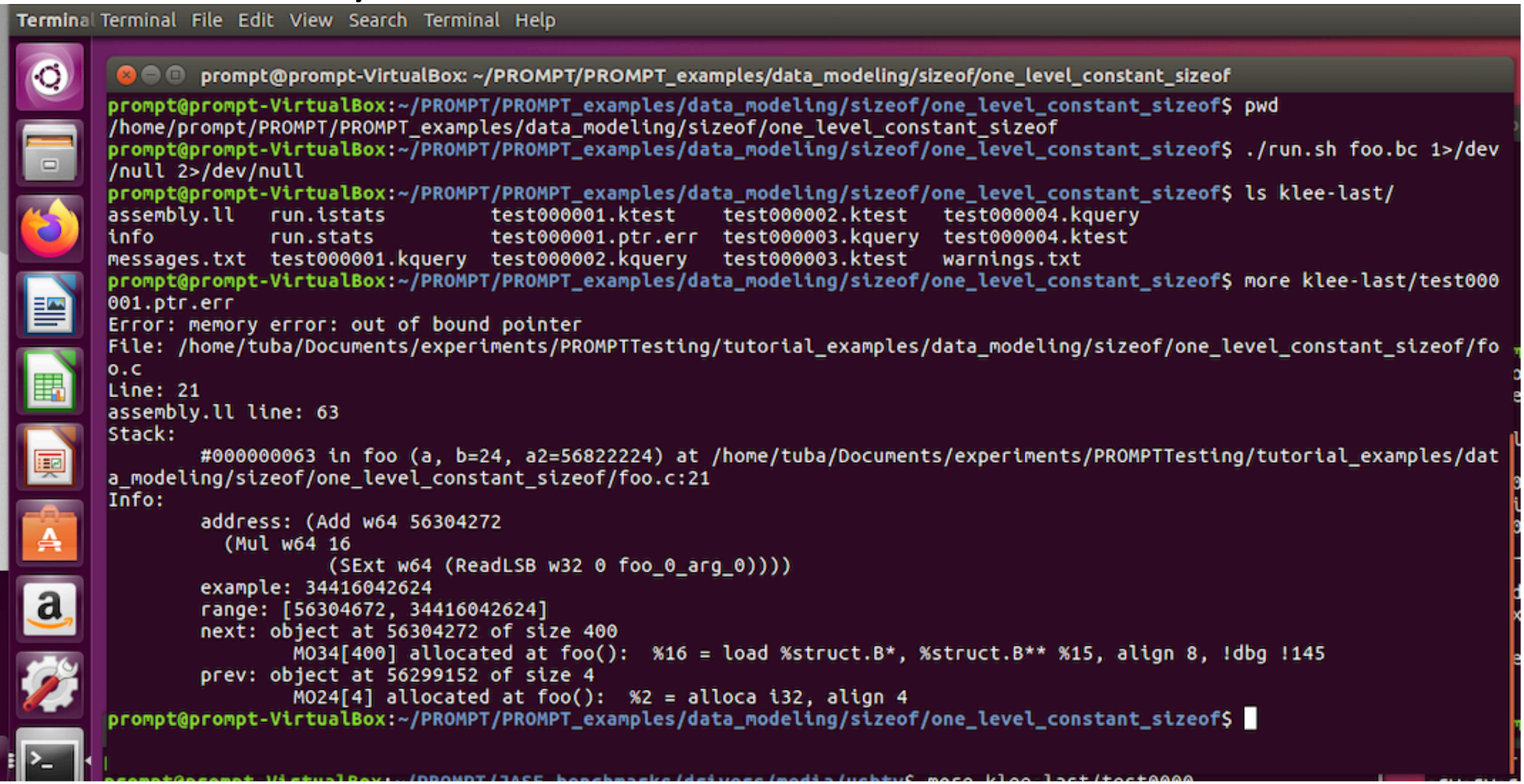
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/sizeof/one_level
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 4
No assertion failure at lines L2 and L4.
Memory error at L6

Let’s try it on the VM!



Specifying data constraints for fields of an embedded struct type

PROSE API MODEL

```
global settings:
data models:

   (x > 24) where x is foo arg 0;

   (x = 24) where x is foo arg 1;

   (25 = w) where w is sizeof A field 2;

function models:
lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
struct A {
 int a;
 int c;
 struct B *b;
};
struct C {
 char c1;
 struct A c2;
};
```

```
void foo(int a, int b, struct C
*c) {
        L1: struct A *a2 = &c->c2;
        L2: if (a <= 24)
        L3:     assert(0);
        L4: if (b != 24)
        L5:     assert(0);
        L6: a2->b[b].b1=4;
        L7: a2->b[a].b1=7;
}
```

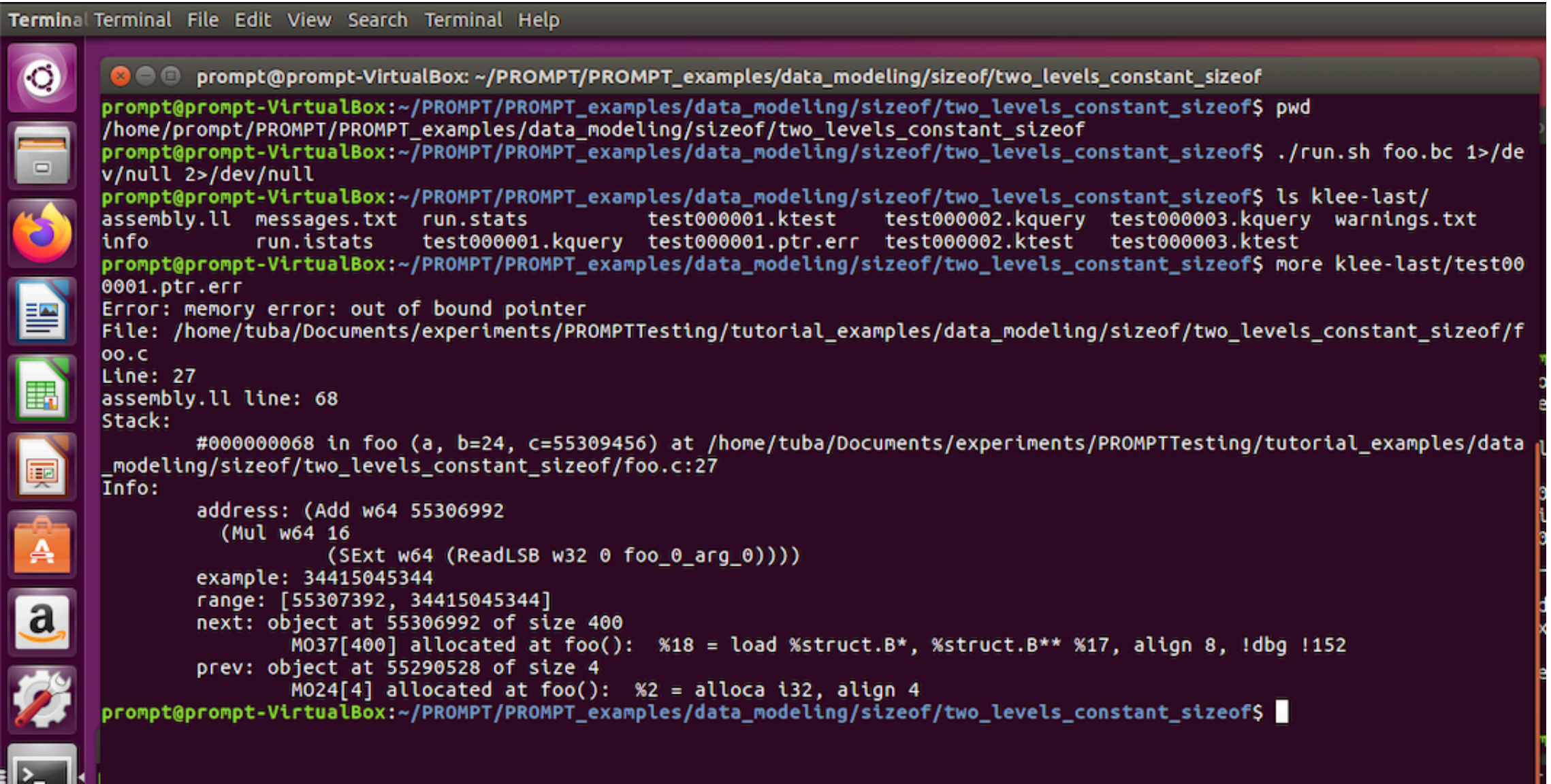
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd data_modeling/sizeof/two_level
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 3
No assertion failures at lines L3 and L5.
Memory error at line L7

Let’s try it on the VM!



Modeling with PROSE

- Global settings
- Data modeling
- **Function modeling**
- Lifecycle modeling

Specifying functions that perform memory allocations and return the address

PROSE API MODEL

```
global settings:
data models:
function models:
  alloc bar sizearg 2
     initzero false symbolize false;
lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
char *bar(int a, int b, int c);

void foo(int *a, int b, int c) {
      L1: c = 10;
      L2: char *n = bar(a,b,c);
      L3: n[9] = 'A';
      L4: n[10] = 'B';            }
```

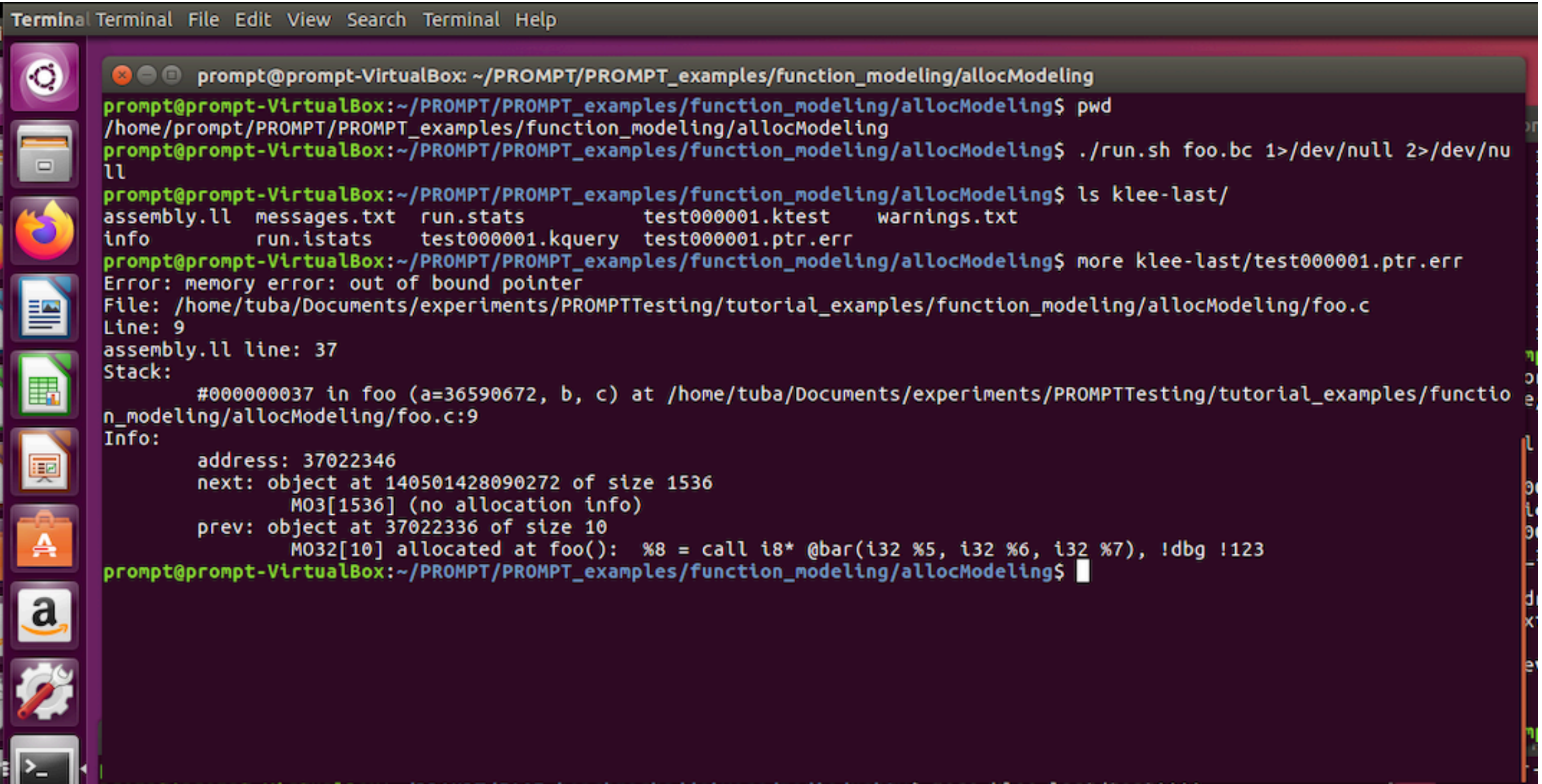
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd function_modeling/allocModeling
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: 1
No memory error at line L3
Memory error at line L4

Let’s try it on the VM!



Specifying functions that perform memory allocations and store the address in an argument

PROSE API MODEL

```
global settings:
data models:
function models:
  alloc bar initzero false
       symbolize false
       memreturn true
       destarg 3;
lifecycle model:
       entry-point foo
```

COMPONENT UNDER ANALYSIS

```
char *bar(int a, int b, int c, char **d);

void foo(int a, int b, int c) {
        L1: char *n, *m;
        L2: c = 10;
        L3: m = bar(a,b,c,&n);
        L4: assert(n == m);
        L5: n[0] = 'A';
        L6: n[1] = 'B';                    }
```

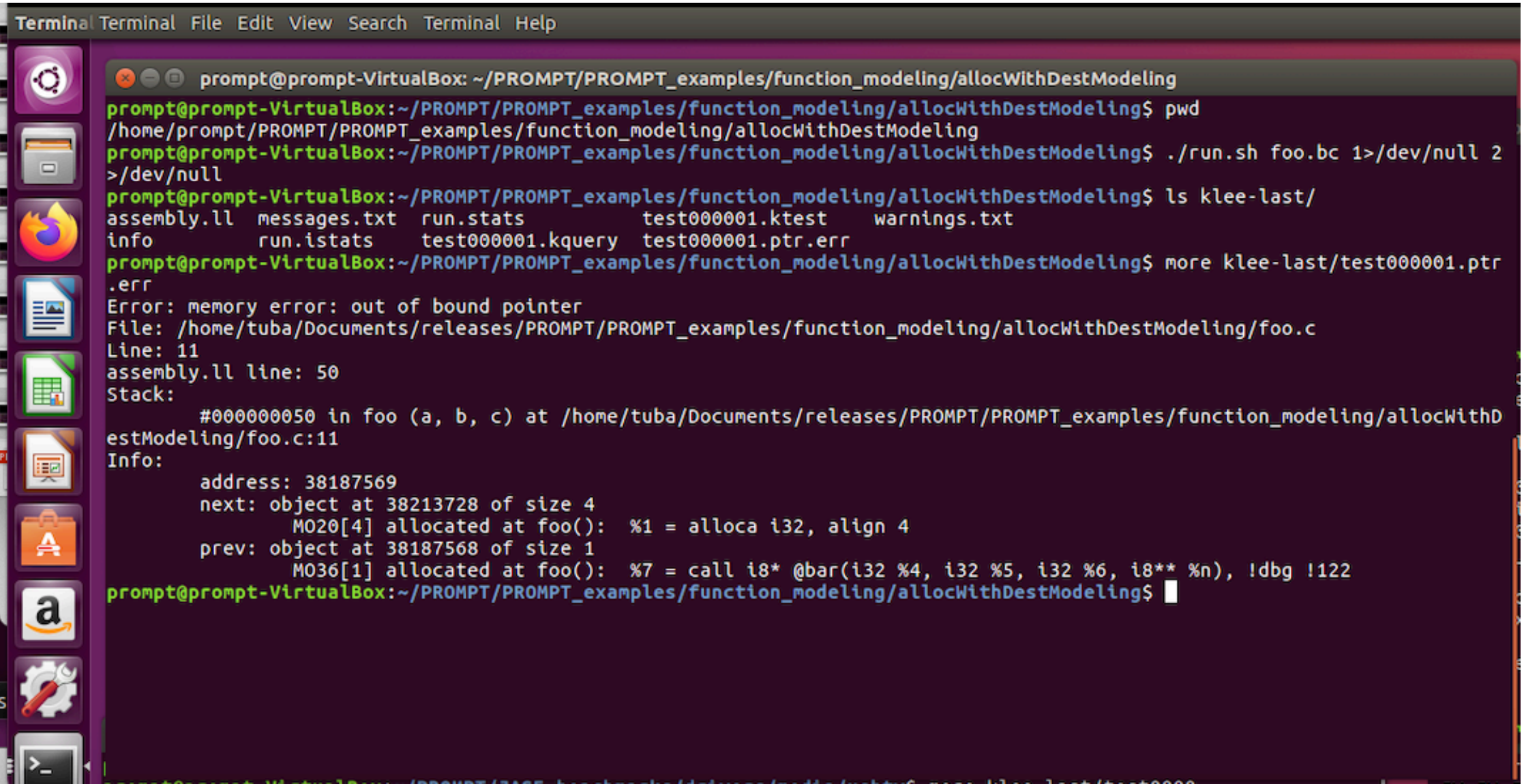
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd function_modeling/allocWithDestModeling
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 1
No assertion failure at line L4
No memory error at line L5
Memory error at line L6

Let’s try it on the VM!



Specifying functions that perform memory deallocation

PROSE API MODEL

```
global settings:
data models:
function models:
  alloc bar sizearg 2
        initzero false
        symbolize false;
  free zot memarg 1;
lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
char *bar(int *a, int *b, int c);
void zot(char *d, char *e);

void foo(int *a, int *b, int c) {
       L1: char *m,*n;
       L2: c = 10;
       L3: n = bar(a,b,c);
       L4: zot(m,n); // first free
       L5: zot(m,n);   // double-free    }
```

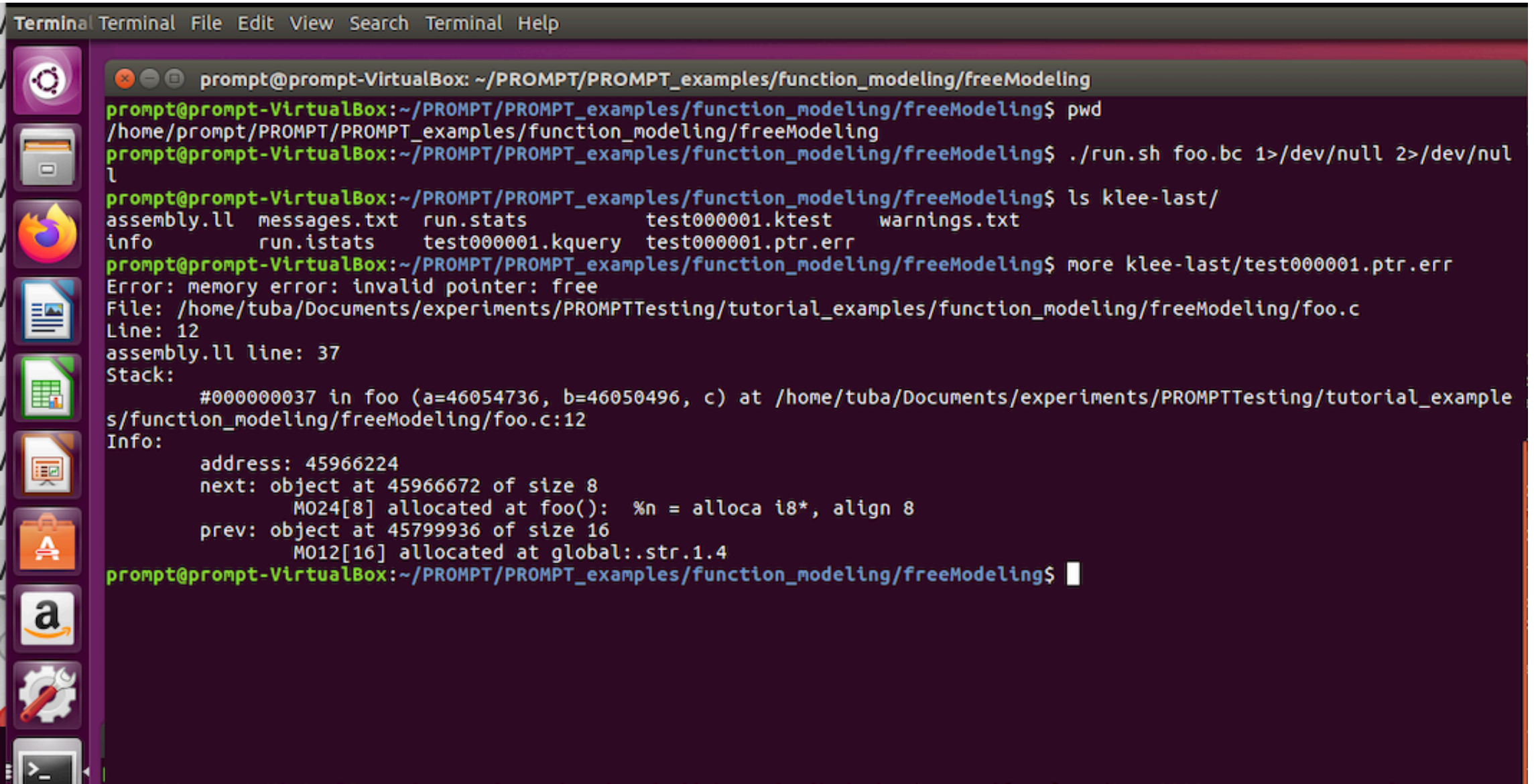
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd function_modeling/freeModeling
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: 1
No memory error at line L4
Memory error at line L5

Let’s try it on the VM!



Specifying functions that are modeled by symbolizing their return value (side-effect free)

PROSE API MODEL

```
global settings:
data models:
function models:
        returnonly bar;
lifecycle model:
        entry-point foo
```

COMPONENT UNDER ANALYSIS

```
int bar(int *a) {
        *a = 5;
        return 0;
}
```

```
void foo(int *a) {
        L1: int y = bar(a);
        L2: if (*a != 5)
        L3:     assert(0);
        L4: if (y)
        L5:     assert(0);

}
```

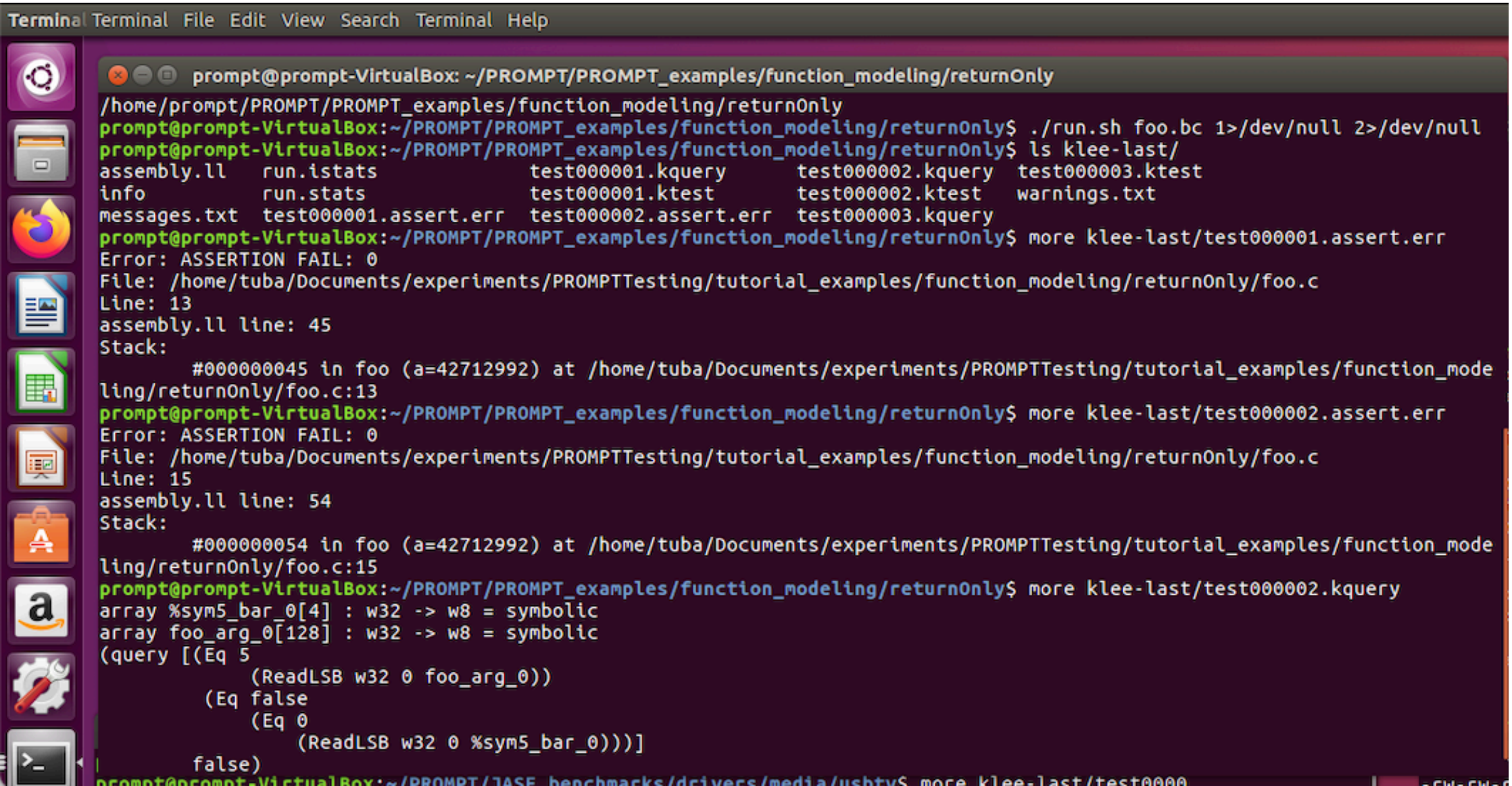
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd function_modeling/returnOnly
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: 3
Assertion failure at lines L3 and L5

Let’s try it on the VM!



Specifying returnonly modeling and havocing of certain function arguments

PROSE API MODEL

```
global settings:
  skip havocing singletons on;
data models:
  singleton A;
function models:
  returnonly bar;
  havoc args 0, 2 of bar;
  havoc args 0 of zot;
lifecycle model:
  entry-point foo
```

COMPONENT UNDER ANALYSIS

```
int zot(struct A *a); int bar(int *a, int *b, int *c);
void foo(int *a, int *b, int *c, struct A *d) {
        L1: *a = *b = *c = d->a = 5;
        L2: zot(d);
        L3: bar(a, b, c);
        L4: if (*a != 5)
        L5:   assert(0);
        L6: if (*b != 5)
        L7:   assert(0);
        L8: if (*c != 5)
        L9:   assert(0);
        L10: if (d->a != 5)
        L11:   assert(0);                                    }
```

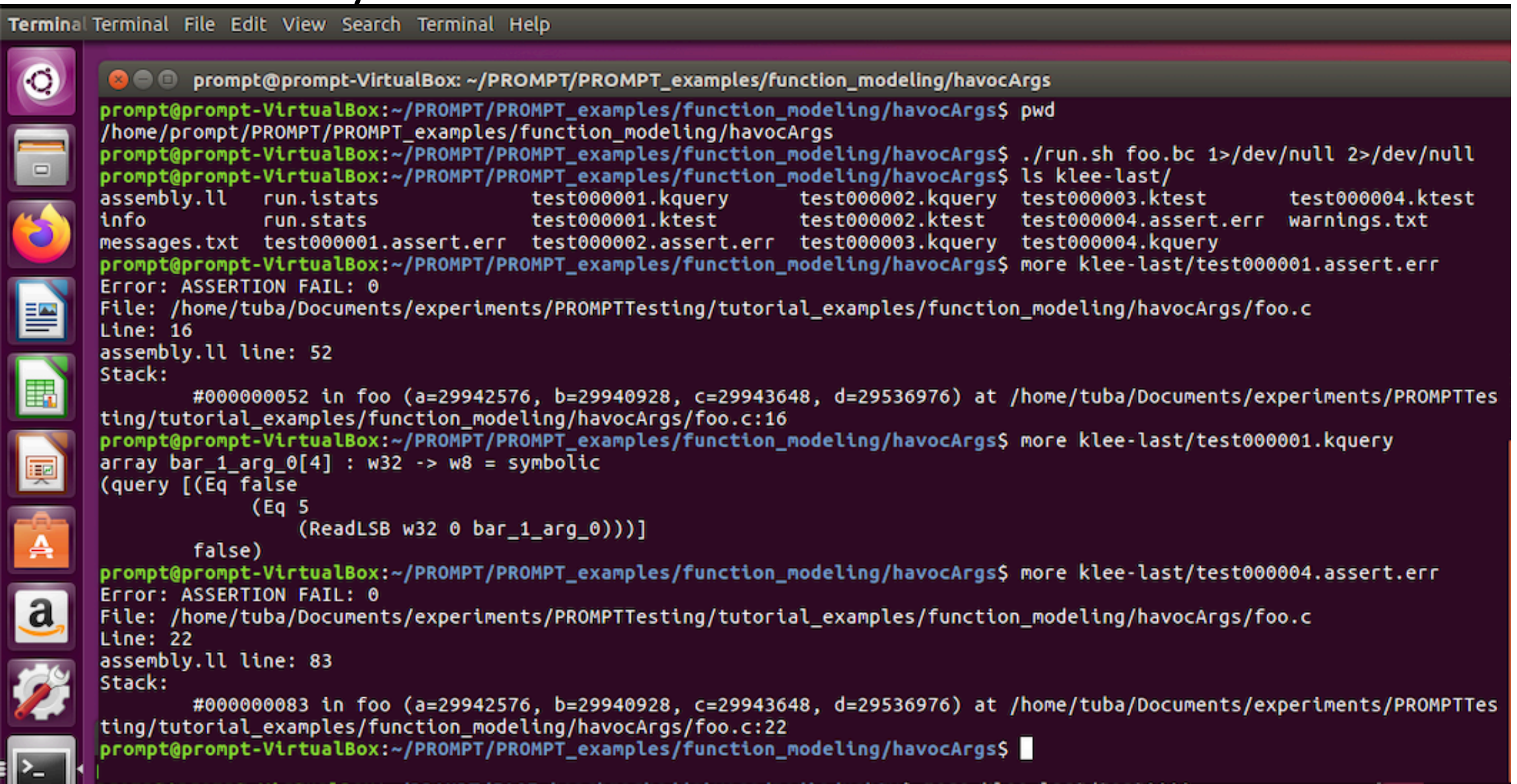
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd function_modeling/havocArgs
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 4
Assertions at lines L5, L9, and L11 reached
Assertion at line L7 not reached

Let’s try it on the VM!



Specifying functions modeled by other functions

PROSE API MODEL

```
global settings:
data models:
function models:
        bar modeled by zot;
lifecycle model:
        entry-point foo
```

COMPONENT UNDER ANALYSIS

```
void bar(int *a);
void zot(int *a) { *a = 5;}
void foo(int *a) {
        L1: bar(a);
        L2: if (*a != 5)
        L3:    assert(0);
}
```

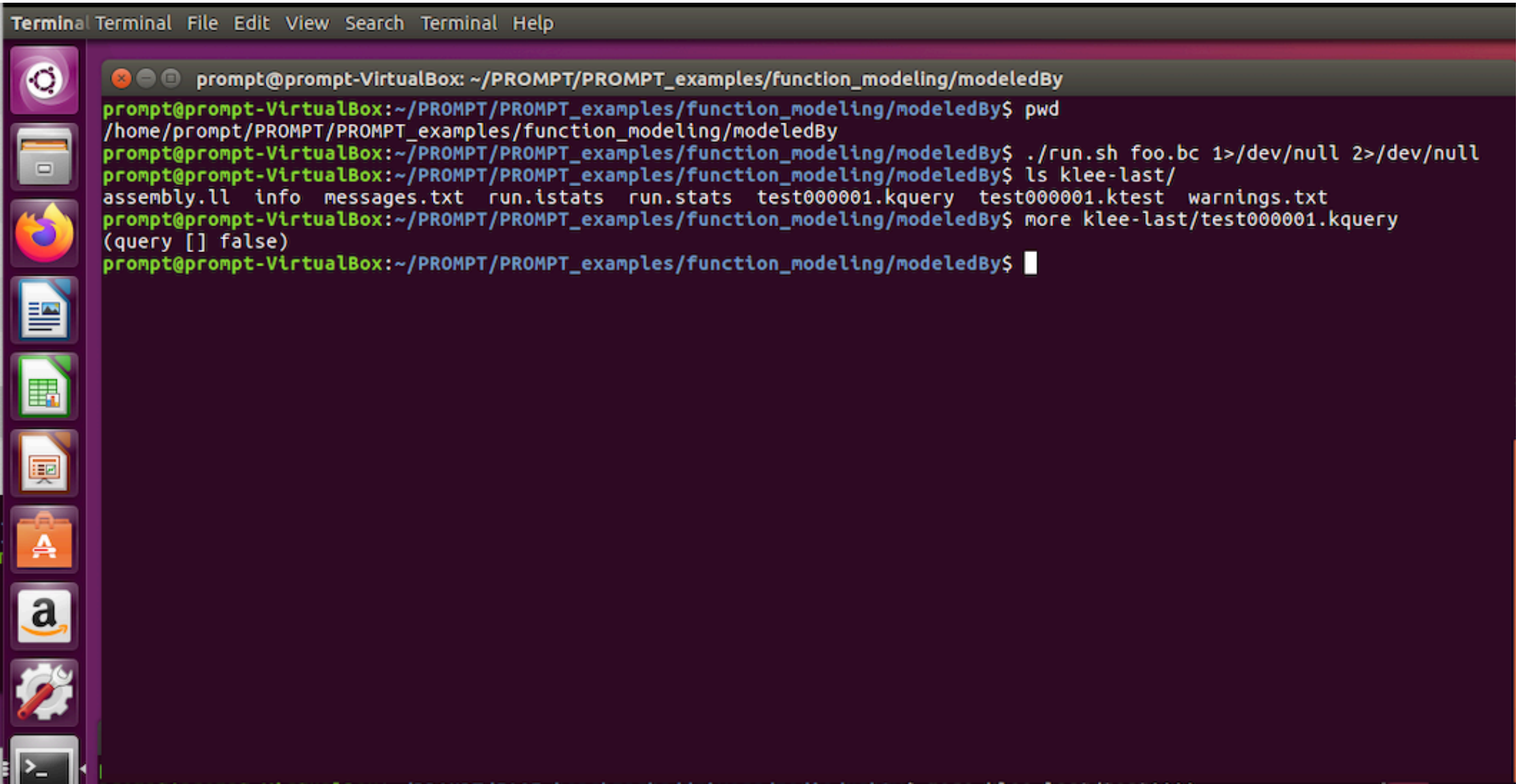
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd function_modeling/modeledBy
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

# of paths: 1
No assertion failure at line L3

Let’s try it on the VM!



Modeling with PROSE

- Global settings
- Data modeling
- Function modeling
- **Lifecycle modeling**

Specifying sequential composition with success values

PROSE API MODEL

```
global settings:
data models:
       singleton A;
function models:
lifecycle model:
         foo[1] ; bar
```

COMPONENT UNDER ANALYSIS

```
struct A {int a;};

int foo(struct A *b){
    if (b->a > 0)
       return 0;
    else return 1;
}
```

```
int bar(struct A *c) {

L1: if (c->a > 0)
L2:     assert(0);
L3: if (c->a == -1)
L4:     assert(0);
L5: return 0;
}
```

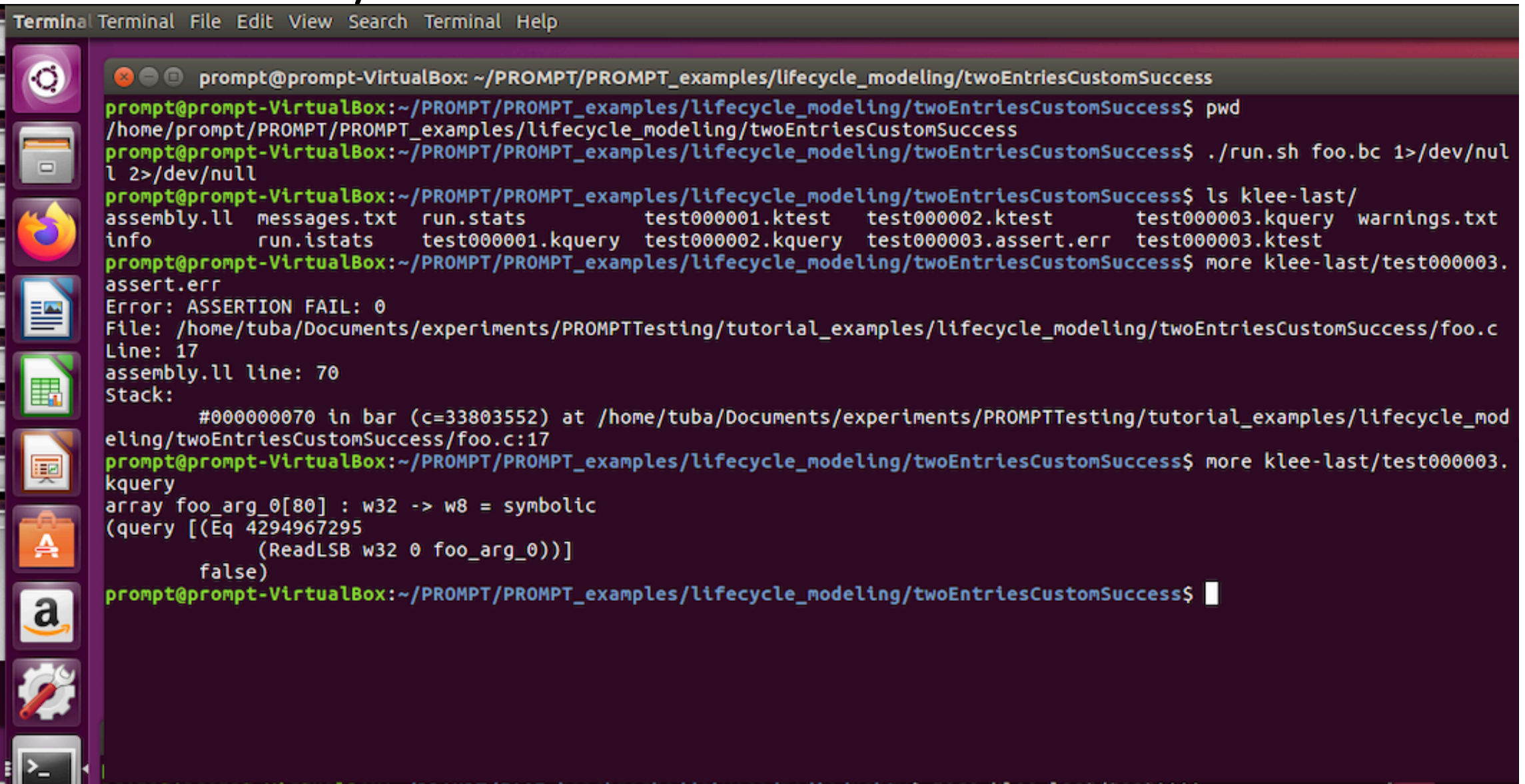
STEPS:

$ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
$ cd PROMPT_examples
$ cd lifecycle_modeling/twoEntriesCustomSuccess
$ ./run.sh foo.bc 2>&1 | tee o.txt
$ ls –l klee-last/

PROMPT RESULTS:

of paths: 3
No assertion error at line L2
Assertion error at line L4

Let’s try it on the VM!



PART III: Real-World Case Studies

Outline for Part III

- openSSL: A Cryptography Library
  - The Heartbleed vulnerability CVE-2014-0160
- BlueZ: A Bluetooth stack
  - A BlueBorne vulnerability CVE-2017-1000251
- Linux kernel
  - A use-after-free in the usbtv driver CVE-2017-17975

Heartbleed - CVE-2014-0160

- A memory out of bounds read error that can be exploited to read secret data from the server or from the client that is participating in a TLS session
  - High severity, leakage of secret data
  - Does not leave any trace in the logs when exploited
- Can be exploited when a specific feature of the TLS protocol is used
  - heartbeats keep connections alive
  - We have used openSSL 1.0.1c to detect the vulnerability
  - See https://heartbleed.com/ for more info
- https://github.com/sysrel/PROMPT/tree/master/caseStudies/openssl/heartbleed
- Directory on the VM
  - $ cd /home/prompt/PROMPT/caseStudies/openssl/heartbleed
- How to run
  - $ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
  - $ ./run.sh 2>/dev/null 1>/dev/null
- Check the result
  - $ more klee-last/test000005.ptr.error

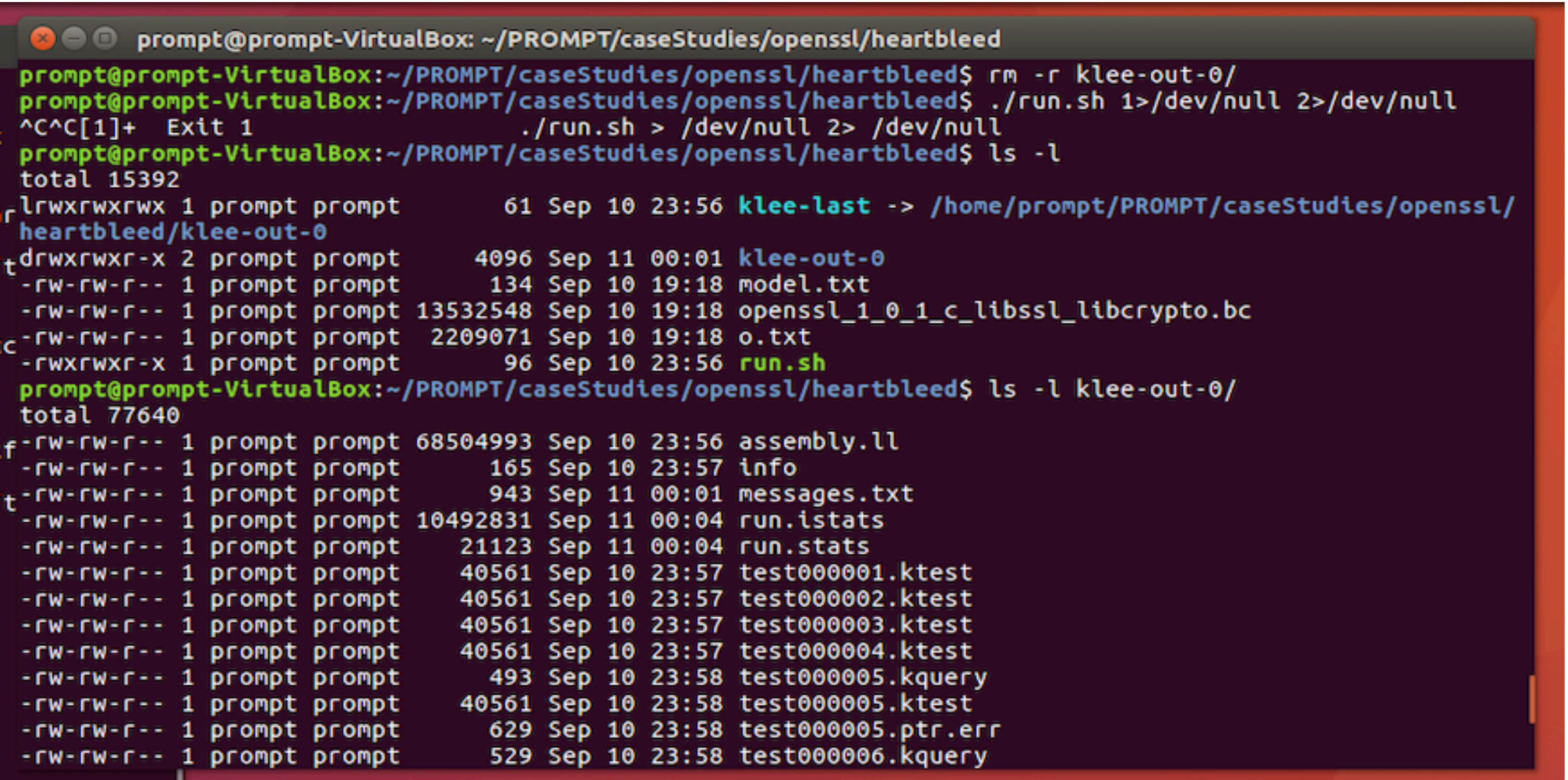
COMPONENT UNDER ANALYSIS

```
Int tls1_process_heartbeat(SSL *s) { …
  if (s->msg_callback)
          s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
               &s->s3->rrec.data[0], s->s3->rrec.length,
               s, s->msg_callback_arg);


 if (hbtype == TLS1_HB_REQUEST) { …
   /* Allocate memory for the response, size is 1 byte
    * message type, plus 2 bytes payload length, plus
    * payload, plus padding
    */
   buffer = OPENSSL_malloc(1 + 2 + payload + padding);
   bp = buffer;

   /* Enter response type, length and copy payload */
   *bp++ = TLS1_HB_RESPONSE;
   s2n(payload, bp);
   memcpy(bp, pl, payload);
   bp += payload;
   /* Random padding */
   RAND_pseudo_bytes(bp, padding);

   r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3   +
payload + padding);
```

PROSE API MODEL

```
global settings:
        array size 2;  // size of pl buffer
        init funcptrs to null on;

data models:

function models:

lifecycle model:

        entry-point tls1_process_heartbeat
```

Let’s try it on the VM!



PROMPT RESULTS:

```
Error: memory error: out of bound pointer
File: /home/tuba/Documents/releases/PROMPT/runtime/Intrinsic/memcpy.c
Line: 17
assembly.ll line: 383044
Stack:
        #000383044 in memcpy (destaddr=330906419, srcaddr=277348915, len) at
/home/tuba/Documents/releases/PROMPT/runt
ime/Intrinsic/memcpy.c:17
        #100353745 in tls1_process_heartbeat (s=330210224) at
/home/tuba/Documents/tools/OPENSSL/openssl-OpenSSL_1_0_1
c/ssl/t1_lib.c:2469
Info:
        address: 277348915
        next: object at 323117216 of size 13
                MO50[13] allocated at global:.str.4.2451
        prev: object at 277348912 of size 16
                MO8691[16] allocated at tls1_process_heartbeat():  %4 = load i8*,
i8** %3, align 8, !dbg !250977, !tba
a !250978
```

PROMPT detects the heartbleed vulnerability within a minute!

BlueBorne CVE-2017-1000251

- Bluetooth is a very popular wireless short range communication protocol
- One of the vulnerabilities dubbed as BlueBorne is in BlueZ
  - BlueZ is the Bluetooth stack used in the Linux kernel
  - The vulnerability can be exploited to perform remote code execution when the extended flow specification (EFS) feature is utilized in L2CAP
- https://github.com/sysrel/PROMPT/tree/master/JASE_benchmarks/bluez/l2cap_config_rsp
- Directory on the VM
  - $ cd /home/prompt/PROMPT/JASE_benchmarks/bluez/l2cap_config_rsp
- How to run
  - $ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
  - $ ./run.sh 2>/dev/null 1>/dev/null
- Check the result
  - $ more klee-last/test000069.ptr.error

BlueBorne CVE-2017-1000251 in a nutshell

```
static inline int l2cap_config_rsp(struct l2cap_conn *conn,
struct l2cap_cmd_hdr *cmd, u8 *data)
{   …
  switch (result) { …
    case L2CAP_CONF_PENDING:
      set_bit(CONF_REM_CONF_PEND, &chan->conf_state);
      if (test_bit(CONF_LOC_CONF_PEND, &chan->conf_state)) {
         char buf[64]; // the buffer involved in the overflow
         len = l2cap_parse_conf_rsp(chan, rsp->data,
         len, buf, &result);
```

```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan,
void *rsp, int len, void *data, u16 *result)
{
  struct l2cap_conf_req *req = data;
  void *ptr = req->data; // points into the buffer
  while (len >= L2CAP_CONF_OPT_SIZE) {      // len is not limited
    len -= l2cap_get_conf_opt(&rsp, &type …);
    switch (type) {
      case L2CAP_CONF_MTU:
        l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
        break;
}
```

```
// No checks on the buffer size!

static void l2cap_add_conf_opt(void **ptr,
u8 type, u8 len, unsigned long val)
{
        struct l2cap_conf_opt *opt = *ptr;
        …// write to the buffer
        *ptr += L2CAP_CONF_OPT_SIZE + len;
}
```

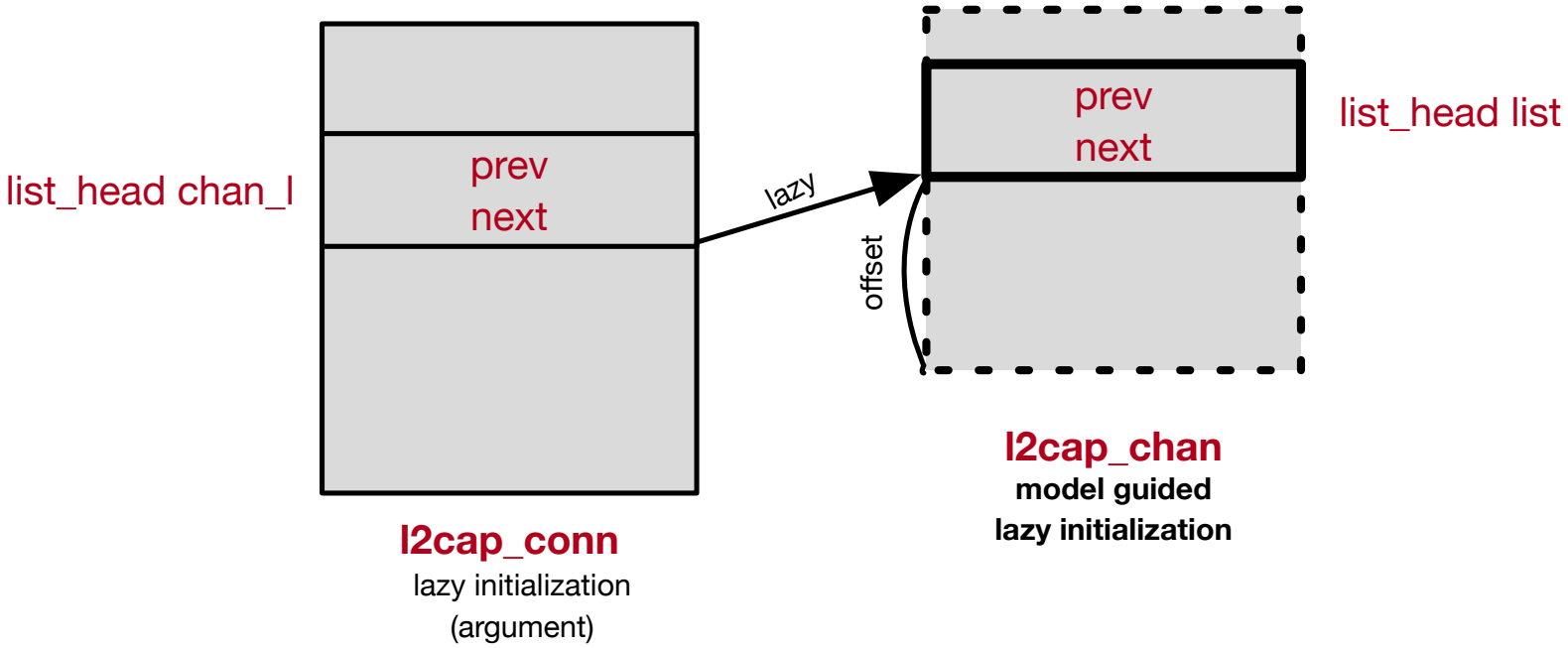
PROSE API MODEL

```
global settings:

        array size 128;

        model funcs with asm off;

        symbolize inline asm on;

data models: …

function models:  …

lifecycle model: …
```

COMPONENT UNDER ANALYSIS

```
static inline int l2cap_config_rsp(
        struct l2cap_conn *conn,
        struct l2cap_cmd_hdr *cmd,
u8 *data)
{   …
  switch (result) { …
    case L2CAP_CONF_PENDING:
        char buf[64]; // the buffer
involved in the overflow
        len =
l2cap_parse_conf_rsp(chan, rsp-
>data,
        len, buf, &result);

        …
```

Pointer arithmetic using a negative offset



COMPONENT UNDER ANALYSIS

```
static inline int l2cap_config_rsp(
        struct l2cap_conn *conn,
        struct l2cap_cmd_hdr *cmd,
u8 *data)
{   …
  chan = l2cap_get_chan_by_scid(conn, scid);
  if (!chan) return 0;
  switch (result) { …
    case L2CAP_CONF_PENDING:
        char buf[64]; // the buffer
involved in the overflow
        len =
l2cap_parse_conf_rsp(chan, rsp-
>data,
        len, buf, &result);

        …
```

Just tell PROMPT that

type l2cap_chan embeds type list_head

PROSE API MODEL

```
data models:


        type l2cap_chan embeds type list_head;


        (x = l2cap_sock_state_change_cb) where
l2cap_sock_state_change_cb is function,
                x is l2cap_ops field 5;


        (x = l2cap_sock_ready_cb) where
l2cap_sock_ready_cb is function,
                x is l2cap_ops field 6;


        (x = l2cap_sock_suspend_cb) where
l2cap_sock_suspend_cb is function,
                x is l2cap_ops field 9;
```

COMPONENT UNDER ANALYSIS

```
static inline int l2cap_config_rsp(
        struct l2cap_conn *conn,
        struct l2cap_cmd_hdr *cmd,
u8 *data)
{    …
   chan = l2cap_get_chan_by_scid(conn, scid);
   if (!chan) return 0;
   switch (result) { …
     case L2CAP_CONF_PENDING:
        char buf[64]; // the buffer
involved in the overflow
        len =
l2cap_parse_conf_rsp(chan, rsp-
>data,
        len, buf, &result);

        …
```

PROSE API MODEL

```
function models:

        alloc usb_alloc_coherent sizearg 1 initzero true
symbolize false;
        alloc __kmalloc sizearg 0 initzero true
symbolize false;
        alloc vzalloc sizearg 0 initzero true symbolize
false;
        free kfree memarg 0;
        free vfree memarg 0;

returnonly skb_clone;
returnonly kfree_skb;
returnonly l2cap_send_cmd;
returnonly hci_send_cmd;
returnonly l2cap_clear_timer;
       …
lifecycle model:
         entry-point l2cap_config_rsp
```

COMPONENT UNDER ANALYSIS

```
static inline int l2cap_config_rsp(
        struct l2cap_conn *conn,
        struct l2cap_cmd_hdr *cmd,
u8 *data)
{    …
   chan = l2cap_get_chan_by_scid(conn, scid);
   if (!chan) return 0;
   switch (result) { …
     case L2CAP_CONF_PENDING:
          char buf[64];// the buffer
involved in the overflow
          len =
l2cap_parse_conf_rsp(chan, rsp-
>data,
          len, buf, &result);

          …
```

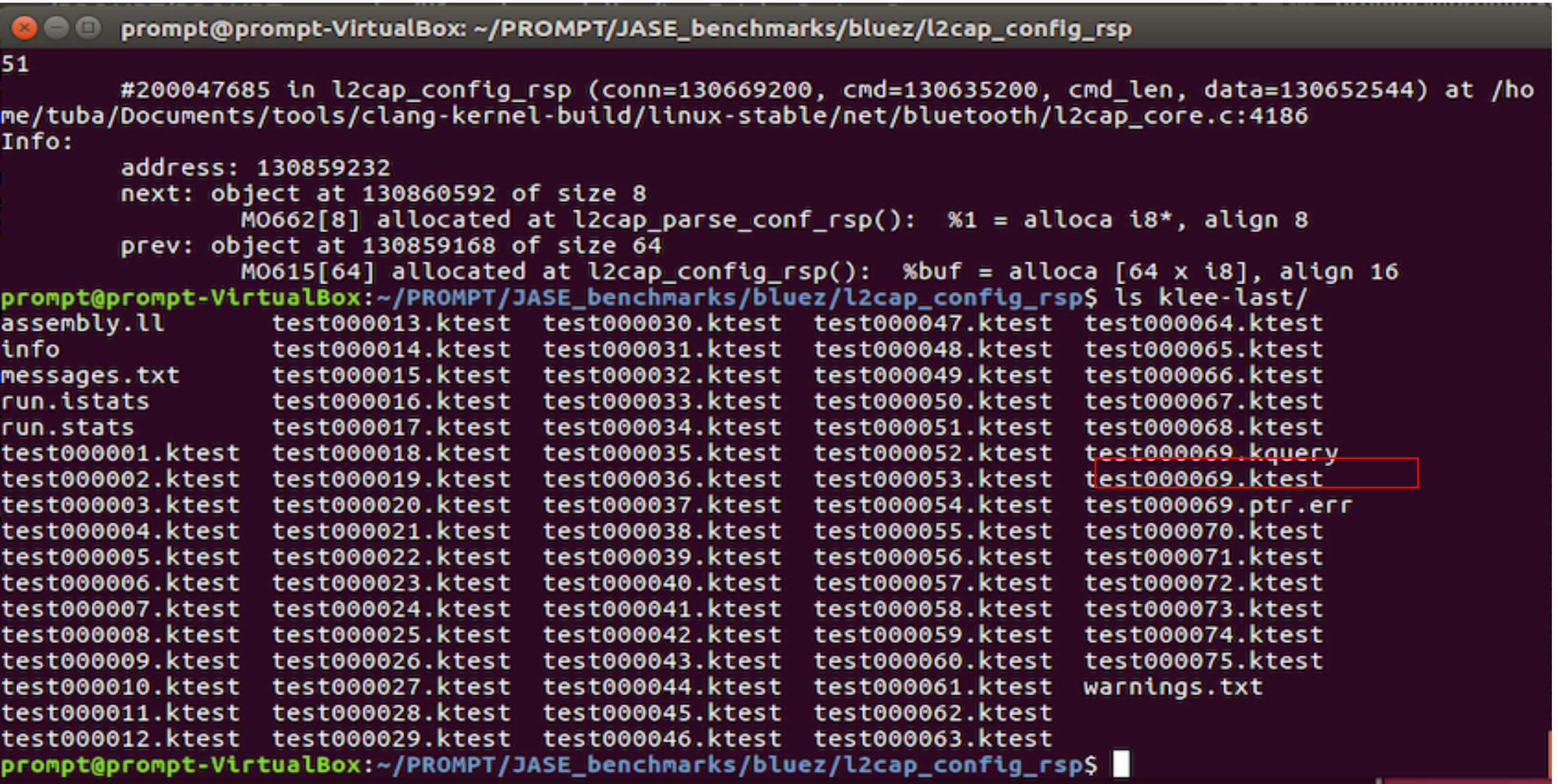
PROSE API MODEL

```
int l2cap_get_conf_opt_PROSE(void **ptr, int *type, int
*olen,
                                        unsigned long *val)

{
        struct l2cap_conf_opt *opt = *ptr;
        int len;

       opt->type = 1;      // L2CAP_CONF_MTU
       opt->len = 2;

        len = L2CAP_CONF_OPT_SIZE + opt->len;
       *ptr += len;

       *type = opt->type;
       *olen = opt->len;
        return len;
}
```

COMPONENT UNDER ANALYSIS

```
static inline int l2cap_config_rsp(
        struct l2cap_conn *conn,
        struct l2cap_cmd_hdr *cmd,
u8 *data)
{   …
  chan = l2cap_get_chan_by_scid(conn, scid);
  if (!chan) return 0;
  switch (result) { …
    case L2CAP_CONF_PENDING:
        char buf[64];// the buffer
involved in the overflow
        len =
l2cap_parse_conf_rsp(chan, rsp-
>data,
        len, buf, &result);

        …
```

Let’s try it on the VM!



PROMPT RESULTS:


```
Error: memory error: out of bound pointer
        File: /home/tuba/Documents/tools/clang-kernel-build/linux-
stable/net/bluetooth/l2cap_core.c
        Line: 2996
        assembly.ll line: 43986
        Stack:
                #000043986 in l2cap_add_conf_opt (ptr=142165200, type=1, len=2,
val=43947) at /home/
        tuba/Documents/tools/clang-kernel-build/linux-
stable/net/bluetooth/l2cap_core.c:2996
                #100048020 in l2cap_parse_conf_rsp (chan=92475136,
rsp=141640710, len, data=14184776
        0, result=141548608) at /home/tuba/Documents/tools/clang-kernel-
build/linux-stable/net/bluet
        ooth/l2cap_core.c:3551
                #200047685 in l2cap_config_rsp (conn=141657120, cmd=141631360,
cmd_len, data=1416407
        04) at /home/tuba/Documents/tools/clang-kernel-build/linux-
stable/net/bluetooth/l2cap_core.c
        :4186
```

PROMPT detects the BlueBorne vulnerability within 5 minutes!

A use-after-free in the usbtv driver CVE-2017-17975

- Implicit updates to the reference counts cause a use-after free and a double free
- Was detected by the MOXCAFE tool
  - Tuba Yavuz. "Detecting Callback Related Deep Vulnerabilities in Linux Device Drivers". SecDev 2019 .
- Required modeling of the registration and deregistration API for the video subsystem
  - a precise modeling of the reference counts and the clean-up callbacks
- https://github.com/sysrel/PROMPT/tree/master/JASE_benchmarks/drivers/media/usbtv
- Directory on the VM
  - $ cd /home/prompt/PROMPT/JASE_benchmarks/drivers/media/usbtv
- How to run
  - $ export PROMPT=/home/prompt/prompt_build_dir/bin/klee
  - $ ./run.sh 2>/dev/null 1>/dev/null
- Check the result
  - $ more klee-last/test000009.ptr.error

A use-after-free CVE-2017-17975 in a nutshell

```
static int usbtv_probe(struct usb_interface *intf,
        const struct usb_device_id *id)
{ …
        ret = usbtv_video_init(usbtv);
        if (ret < 0)
                goto usbtv_video_fail;

        ret = usbtv_audio_init(usbtv);
        if (ret < 0)
                goto usbtv_audio_fail;

usbtv_audio_fail:
        usbtv_video_free(usbtv); // first free

usbtv_video_fail:
        usb_set_intfdata(intf, NULL);
        usb_put_dev(usbtv->udev); // use-after-free
        kfree(usbtv); // double-free

        return ret;
}
```

PROSE API MODEL

```
global settings:

        model funcs with asm off;

data models:

        singleton usb_interface;
        singleton vb2_queue;
        singleton v4l2_device;
        singleton usb_device;
        singleton usbtv;

function models:

        alloc usb_alloc_coherent sizearg 1 initzero true symbolize
false;
        alloc snd_card_new initzero true symbolize true memreturn
false destarg 5;
        alloc snd_pcm_new initzero true symbolize true memreturn
false destarg 5;
        alloc __kmalloc sizearg 0 initzero true symbolize false;
        free kfree memarg 0;


        returnonly dev_err;
        returnonly _dev_info    …
```

COMPONENT UNDER ANALYSIS

```
static int usbtv_probe(struct
usb_interface *intf,
const struct usb_device_id *id)
{ …
 ret = usbtv_video_init(usbtv);
 if (ret < 0)
    goto usbtv_video_fail;
 ret = usbtv_audio_init(usbtv);
 if (ret < 0)
    goto usbtv_audio_fail;
 usbtv_audio_fail:
         usbtv_video_free(usbtv);
 usbtv_video_fail:
         usb_set_intfdata(intf, NULL);
         usb_put_dev(usbtv->udev);
         kfree(usbtv);
         return ret;
}
```

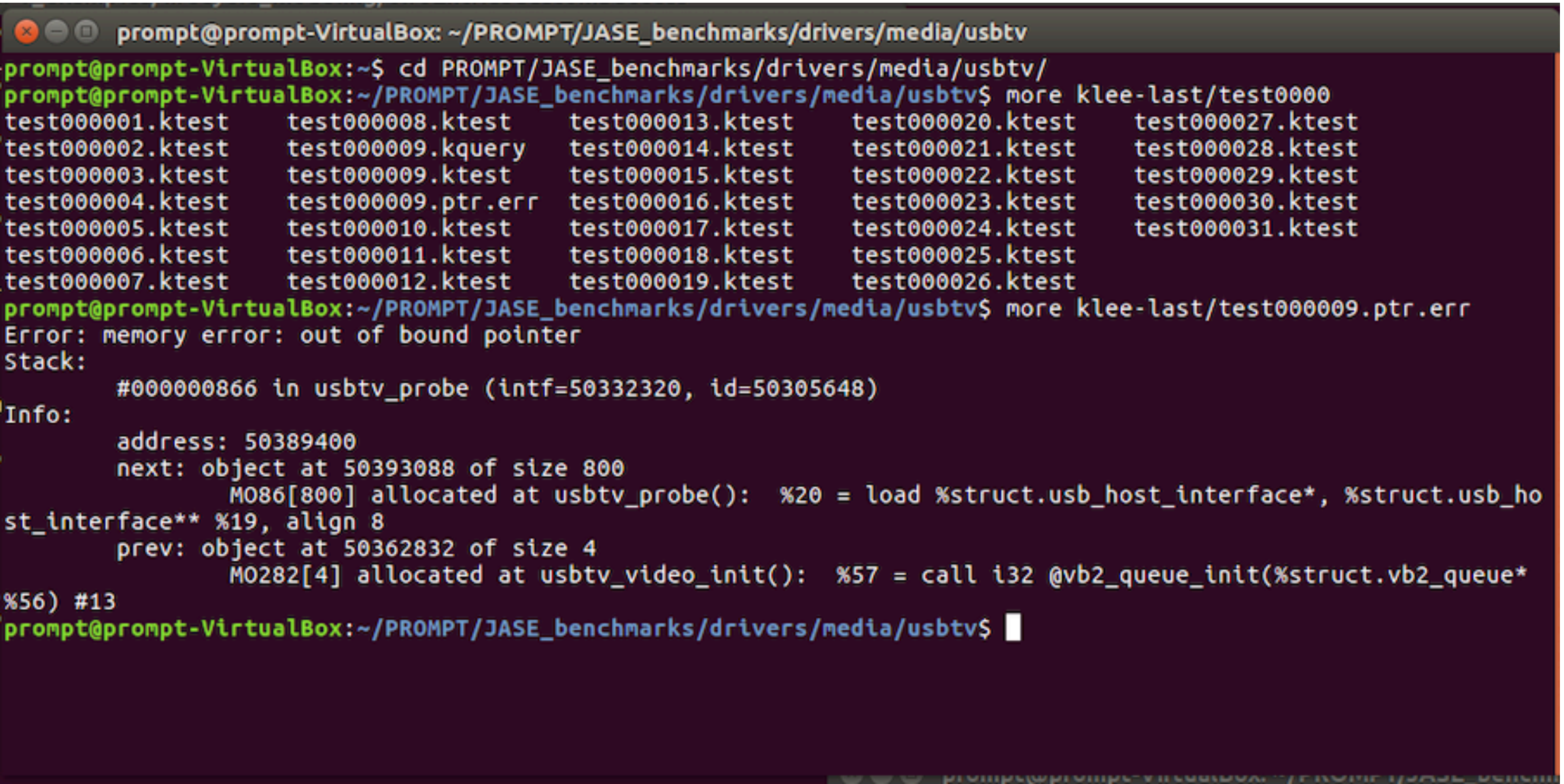
PROSE API MODEL

```
function models: ..
        __video_register_device modeled by __video_register_device_PROSE;

         video_unregister_device modeled by video_unregister_device_PROSE;

         v4l2_device_register modeled by v4l2_device_register_PROSE;

         v4l2_device_unregister modeled by v4l2_device_unregister_PROSE;

         v4l2_device_disconnect modeled by v4l2_device_disconnect_PROSE;

         v4l2_device_put modeled by v4l2_device_put_PROSE;
         v4l2_device_get modeled by v4l2_device_get_PROSE;
         atomic_add_return modeled by atomic_add_return_PROSE;
         atomic_dec modeled by atomic_dec_PROSE;
         atomic_inc modeled by atomic_inc_PROSE;
         v4l2_i2c_subdev_init modeled by v4l2_i2c_subdev_init_PROSE;

lifecycle model:

         usbtv_probe[0] ; usbtv_disconnect
```

COMPONENT UNDER ANALYSIS

```
static int usbtv_probe(struct
usb_interface *intf,
const struct usb_device_id *id)
{ …
 ret = usbtv_video_init(usbtv);
 if (ret < 0)
    goto usbtv_video_fail;
 ret = usbtv_audio_init(usbtv);
 if (ret < 0)
    goto usbtv_audio_fail;
 usbtv_audio_fail:
          usbtv_video_free(usbtv);
 usbtv_video_fail:
          usb_set_intfdata(intf, NULL);
          usb_put_dev(usbtv->udev);
          kfree(usbtv);
          return ret;
}
```

Let’s try it on the VM!



PROMPT RESULTS:

CVE-2017-17975

```
Error: memory error: out of bound pointer
Stack:
        #000000866 in usbtv_probe (intf=43920800, id=43943728)
Info:
        address: 44033848
```

```
; <label>:79                                      ; preds = %50, %78
  %80 = load %struct.usb_interface*, %struct.usb_interface** %2, align 8
  call void @usb_set_intfdata(%struct.usb_interface* %80, i8* null) #13
  %81 = load %struct.usbtv*, %struct.usbtv** %usbtv, align 8
  %82 = getelementptr inbounds %struct.usbtv, %struct.usbtv* %81, i32 0, i32

  %83 = load %struct.usb_device*, %struct.usb_device** %82, align 8
  call void @usb_put_dev(%struct.usb_device* %83) #13
```

klee-last/assembly.ll

```
usbtv_video_fail:
        usb_set_intfdata(intf, NULL);
        usb_put_dev(usbtv->udev); // use-after-free
        kfree(usbtv); // double-free

        return ret;
}
```

PROMPT detects the use-after vulnerability within 1 minute!

Applying PROMPT to your component

- Install PROMPT on your high-end computer
  - The virtual machine instance is just for getting started!
- Generate the bitcode for your project
  - Check out the wllvm tool at https://github.com/travitch/whole-program-llvm
  - You may need to link the bitcode generated for individual libraries using llvm-link
- Start with a simple model
  - Entry-point function_to_analyze
- Expand the model until you achieve your goal
  - Coverage
  - Reaching certain program locations
  - Finding bugs!
- Happy modeling and bug hunting!

Acknowledgements

- This work has been partially funded by
  - National Science Foundation (NSF) awards CNS-1815883 and CNS-1942235
  - Semiconductor Research Corporation (SRC)
- Thanks to Joshua Nelson for working on the parser as an undergraduate researcher

- If you use PROMPT for your work, please cite our paper
  - Tuba Yavuz and Ken (Yihang) Bai. ***Analyzing System Software Components using API Model Guided Symbolic Execution.*** To appear in Automated Software Engineering (27:6).DOI: 10.1007/s10515-020-00276-5

Thank you - Questions?