Fuzzing Class Interfaces for Generating and Running Tests with libFuzzer*

Barnabás Bágyi, Zoltán Porkoláb

*This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

Overview

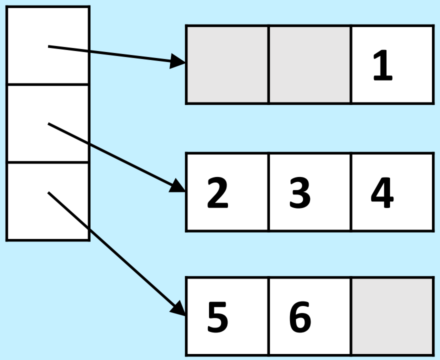
- Deficiencies of the testing ecosystem
- General fuzzing and libFuzzer introduction
- Design of an interface fuzzer
- Case studies

Let's follow the design and testing of a (container) class

The (simplified) container vision

Double ended queue

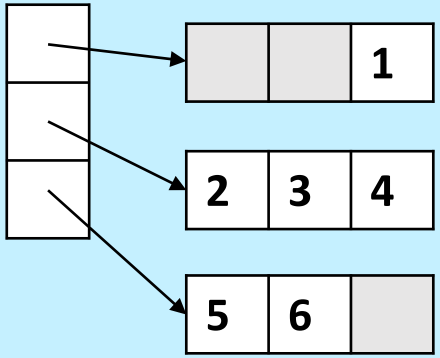
- similar to std::deque
- implemented with a vector of static sized arrays



The (simplified) container vision

Double ended queue

- similar to std::deque
- implemented with a vector of static sized arrays



```
struct my_deque {
    void push_back(int);
    void pop_back();
    int back() const;

    void push_front(int);
    void pop_front();
    int front() const;

    std::size_t size() const;
private:
    // ...
};
```

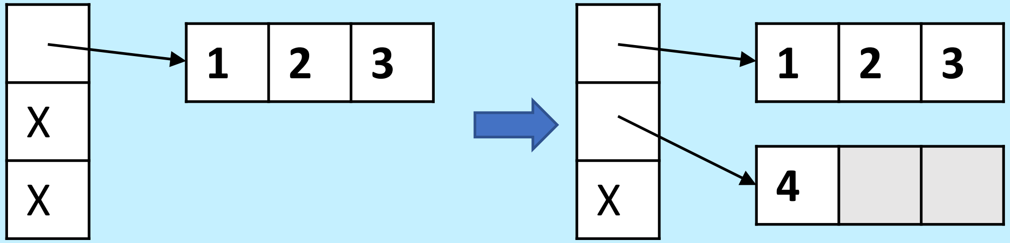
Example Unit Test Case

```
TEST(my_deque_test, push_pop)
{
    my_deque md;
    ASSERT_EQ(md.size(), 0);

    md.push_back(42);
    ASSERT_EQ(md.size(), 1);
    ASSERT_EQ(md.back(), 42);

    md.pop_back();
    ASSERT_EQ(md.size(), 0);
}
```

- Testing only a small part of the software - one unit
- Sequence of method calls and state assertions
- Did we do enough, if we only write this and a similar front test case?

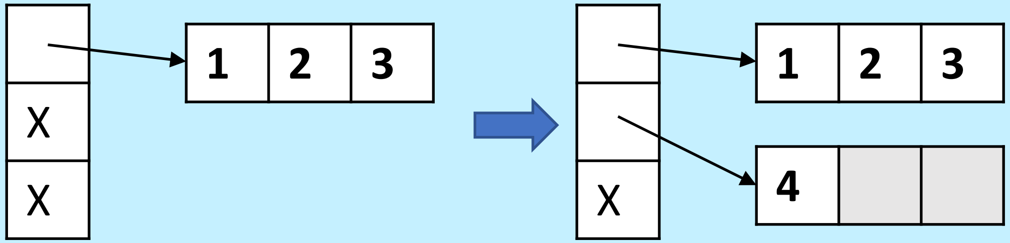
Possible undetected bugs

- Creation of a new array

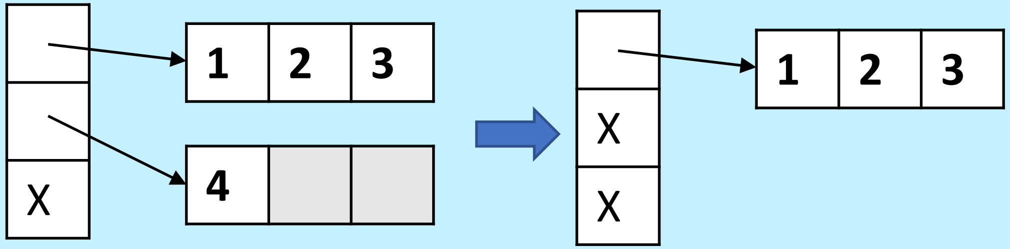


Possible undetected bugs

- Creation of a new array

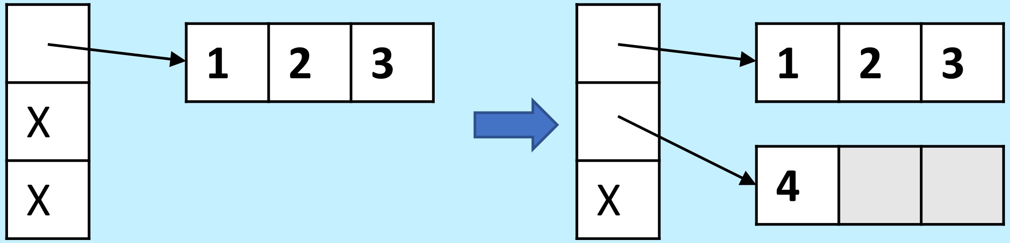


- Destruction of a new array

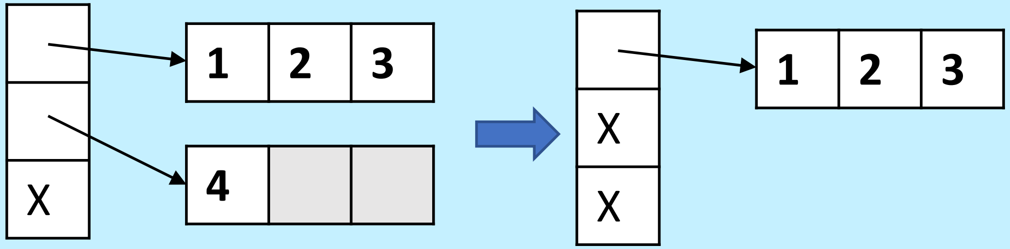


Possible undetected bugs

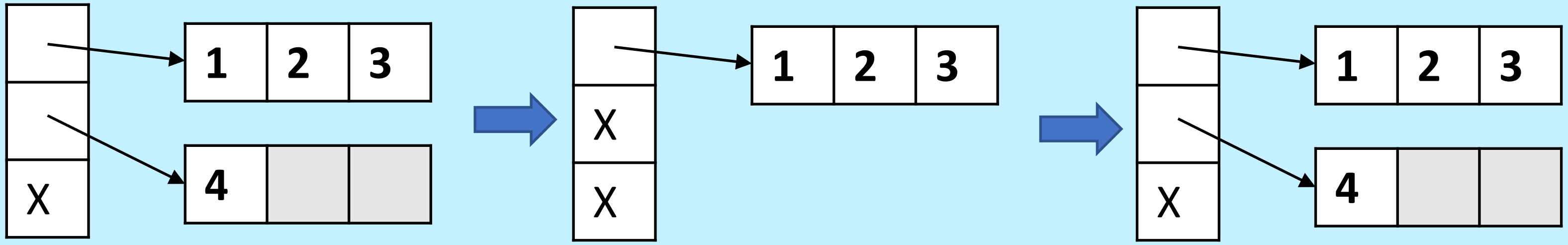
- Creation of a new array



- Destruction of a new array



- Destruction of a new array, then a recreation of it

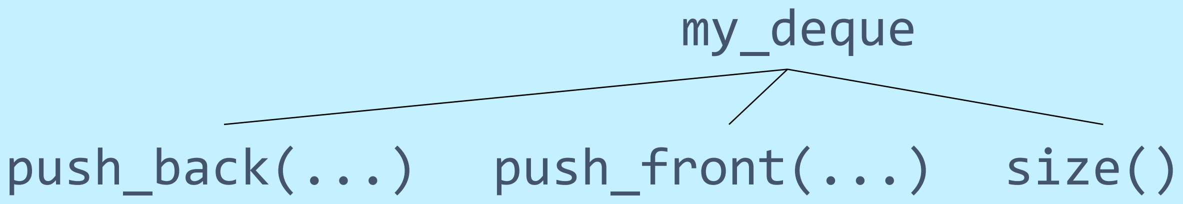


The last one contributes no additional LOC coverage

Too many states

```
my_deque
```

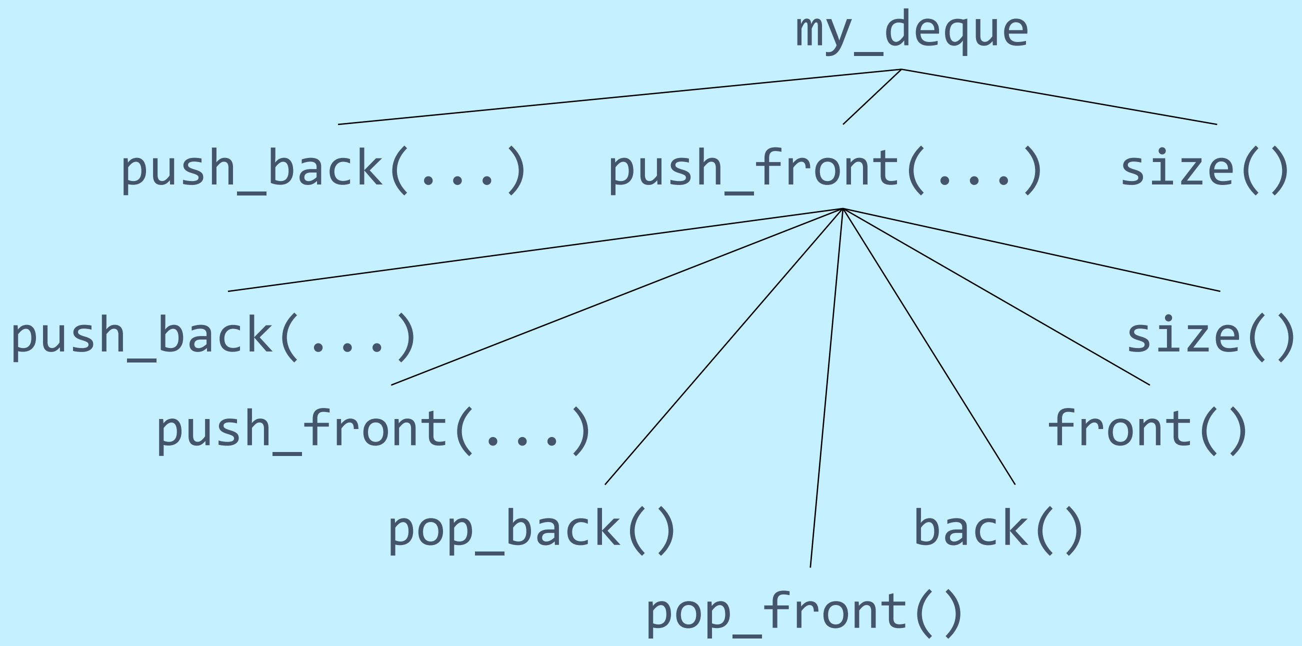
Too many states



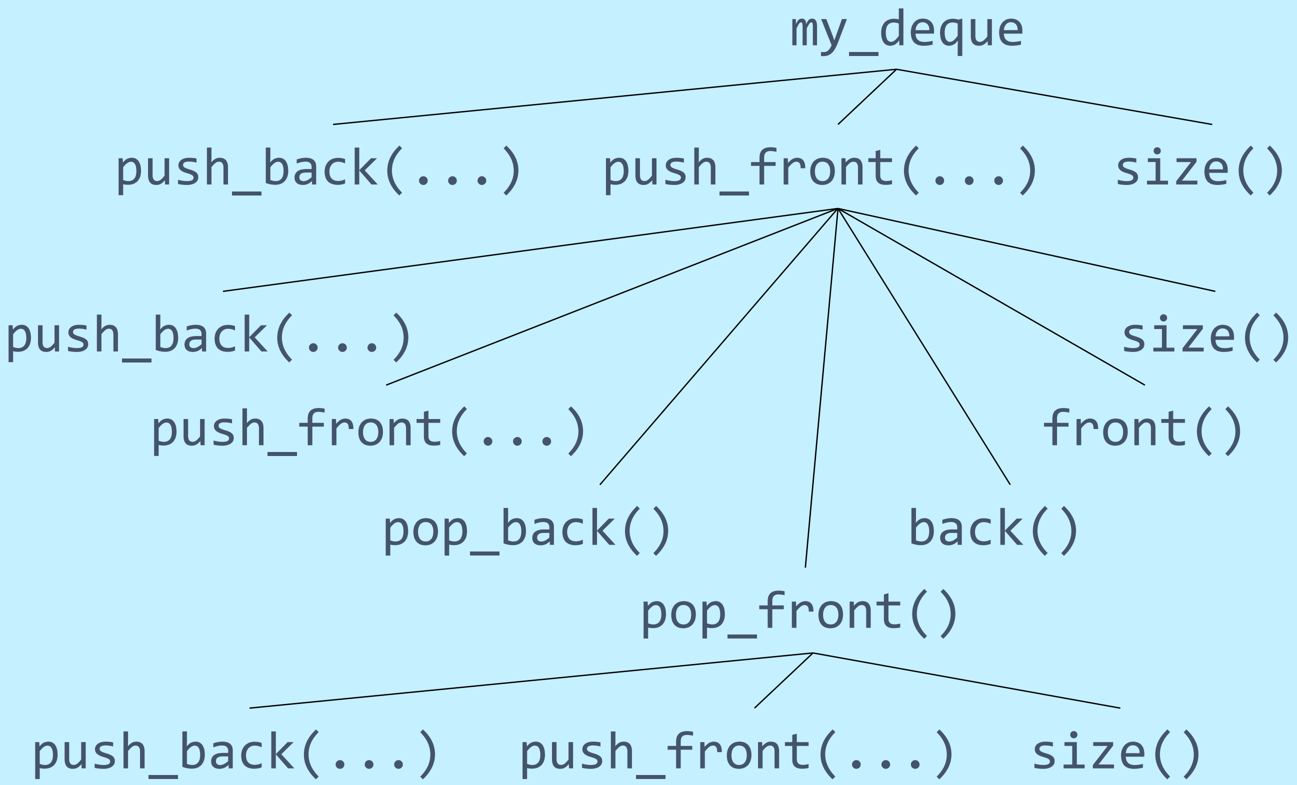
my_deque

push_back(...) push_front(...) size()

Too many states



Too many states



my_deque

push_back(...) push_front(...) size()

push_back(...)

push_front(...)

pop_back()

pop_front()

back()

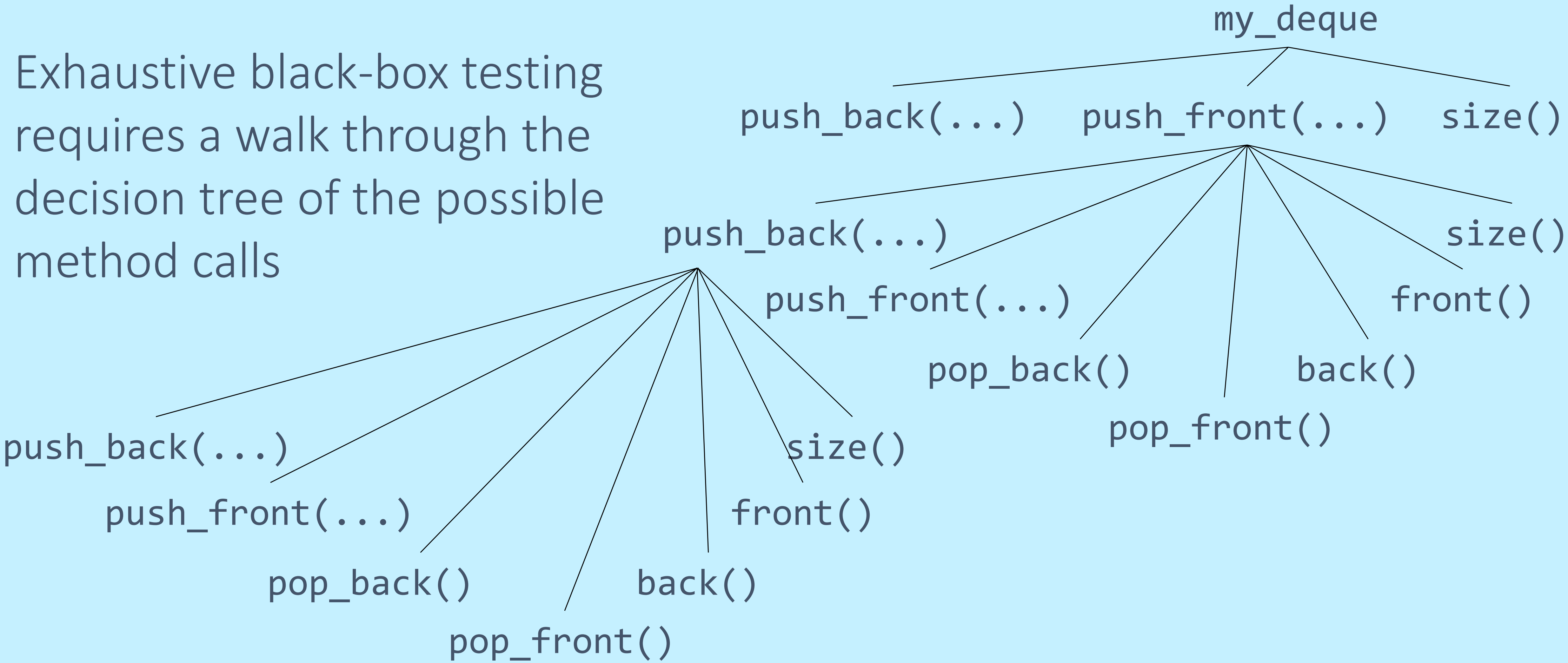
front()

size()

push_back(...) push_front(...) size()

Too many states

- Exhaustive black-box testing requires a walk through the decision tree of the possible method calls



my_deque

push_back(...) push_front(...) size()

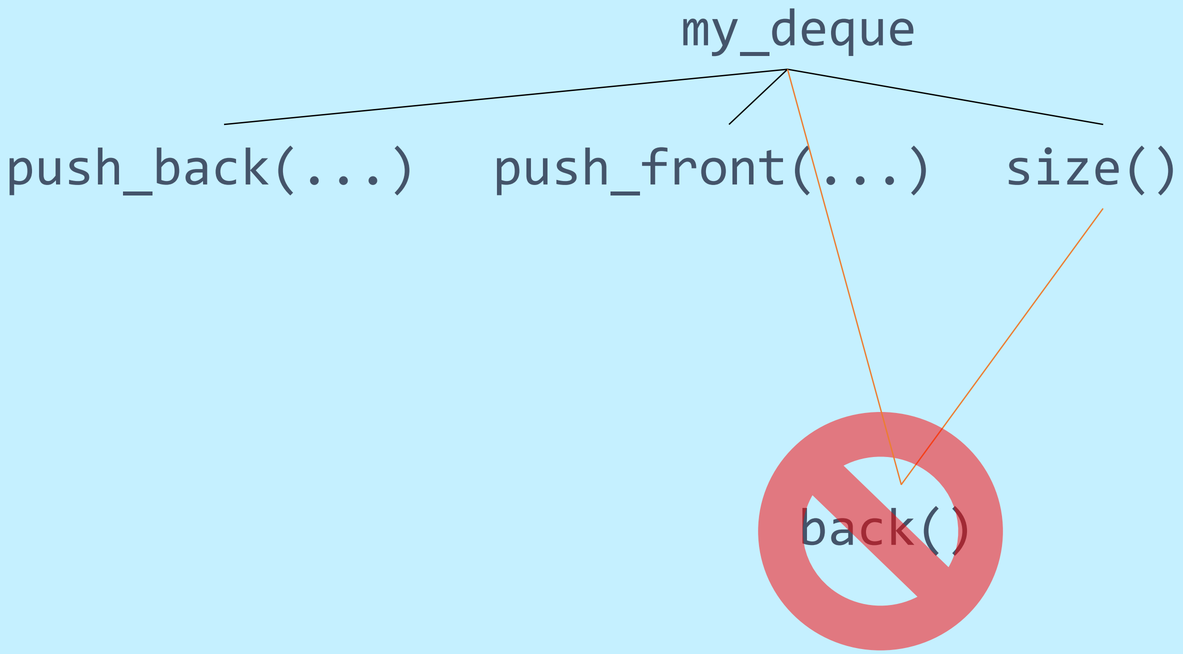
push_back(...)
push_front(...)
pop_back()
pop_front()
back()
front()
size()

push_back(...)
push_front(...)
pop_back()
pop_front()
back()
front()
size()

The sky is the limit!

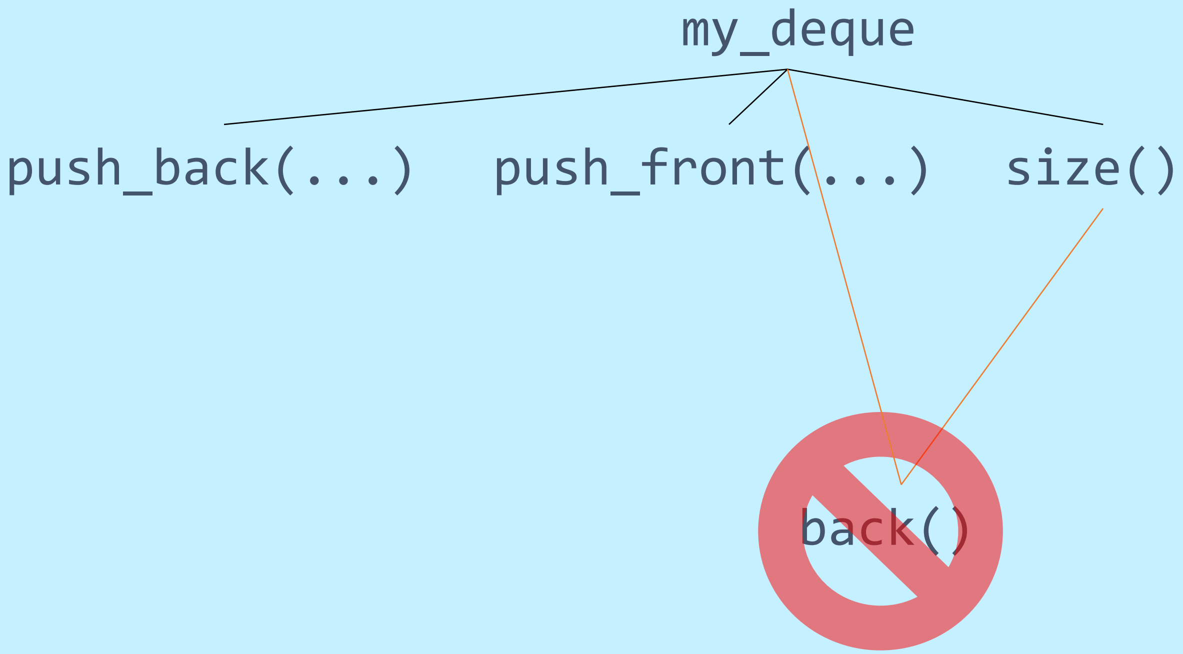
Too many states

- Exhaustive black-box testing requires a walk through the decision tree of the possible method calls
- Still need to pay attention to the preconditions



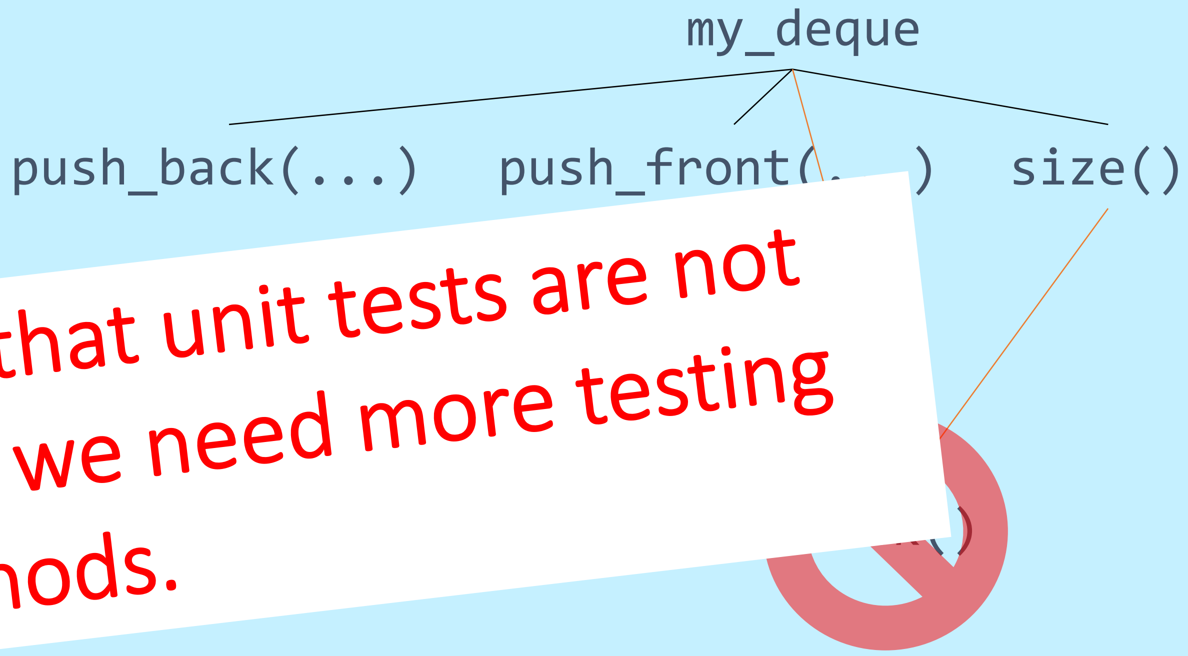
Too many states

- Exhaustive black-box testing requires a walk through the decision tree of the possible method calls
- Still need to pay attention to the preconditions
- Whitebox testing is limited by the imagination of the test developer



Too many states

- Exhaustive black-box testing requires a walk through the decision tree of the possible method c
- St
  th
- Wh
  by imagination of the
  test developer



This is does not mean that unit tests are not worth doing. Only that we need more testing methods.

What we would need

- Automatic generation of test cases based on class interface

```cpp
struct my_deque {
    void push_back(int);
    void pop_back();
    int back() const;

    void push_front(int);
    void pop_front();
    int front() const;

    std::size_t size() const;
private:
    // ...
};
```

```cpp
my_deque md;
md.push_back(12);
md.push_back(35);
```

What we would need

- Automatic generation of test cases based on class interface

```
struct my_deque {
    void push_back(int);
    void pop_back();
    int back() const;

    void push_front(int);
    void pop_front();
    int front() const;

    std::size_t size() const;
private:
    // ...
};
```

```
my_deque md;
md.push_back(12);
md.push_back(35);

my_deque md;
md.push_back(12);
md.pop_front();
```

What we would need

- Automatic generation of test cases based on class interface

```
struct my_deque {
    void push_back(int);
    void pop_back();
    int back() const;

    void push_front(int);
    void pop_front();
    int front() const;

    std::size_t size() const;
private:
    // ...
};
```

```
my_deque md;
md.push_back(12);
md.push_back(35);

my_deque md;
md.push_back(12);
md.pop_front();

my_deque md;
md.pop_front();
md.push_back(12);
```

What we would need

- Automatic generation of test cases based on class interface

```
struct my_deque {
    void push_back(int);
    void pop_back();
    int back() const;

    void push_front(int);
    void pop_front();
    int front() const;

    std::size_t size() const;
private:
    // ...
};
```

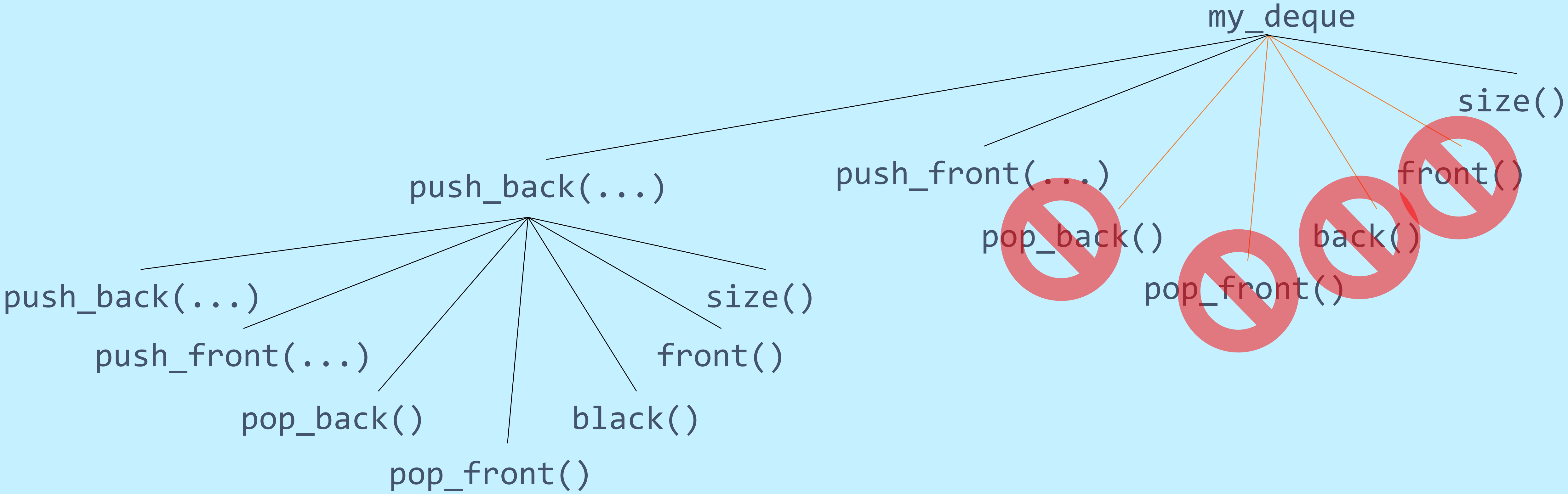
```
my_deque md;
md.push_back(12);
md.push_back(35);

my_deque md;
md.push_back(12);
md.pop_front();
```

🚫

```
my_deque md;
md.pop_front();
md.push_back(12);
```

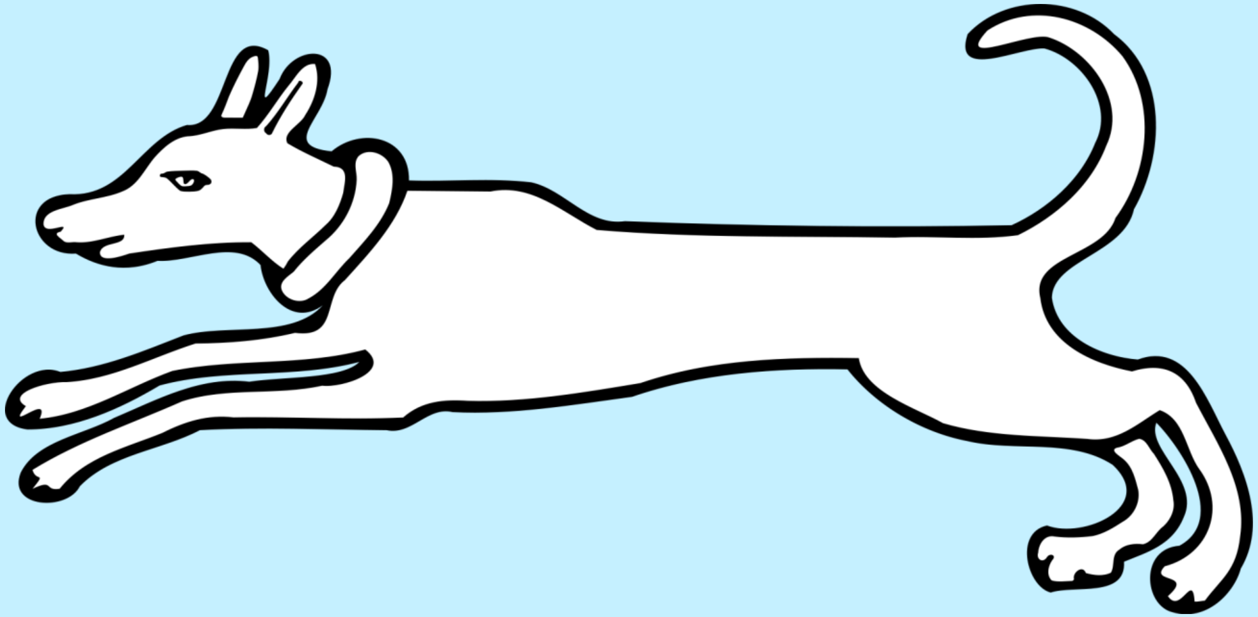
What we would need

- Automatic generation of test cases based on class interface
- Filtering out invalid method calls



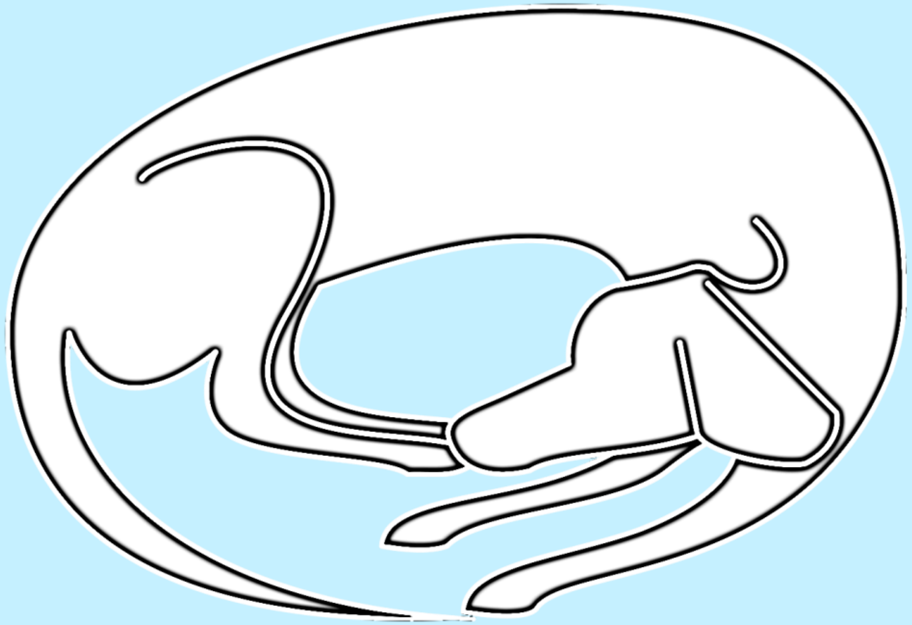
What we would need

- Automatic generation of test cases based on class interface
- Filtering out invalid method calls
- Running test cases on the fly



What we would need

- Automatic generation of test cases based on class interface
- Filtering out invalid method calls
- Running test cases on the fly
- Persisting test cases for later regression testing



What we would need

- Automatic generation of test cases based on class interface
- Filtering out invalid method calls
- Running test cases on the fly
- Persisting test cases for later regression testing
- Filtering out redundant test cases

```
MyDeque md;
MyDeque.size();
```

<

```
MyDeque md;
MyDeque.size();
MyDeque.size();
```

What we would need

- Automatic generation of test cases based on class interface
- Filtering out invalid method calls
- Running test cases on the fly
- Persisting test cases for later regression testing
- Filtering out redundant test cases
- Maximize combined coverage

What we would need

- Automatic generation of test cases based on class interface
- Filtering out invalid method calls
- Running test cases on the fly
- Persisting test cases for later regression testing
- Filtering out redundant test cases
- Maximize combined coverage
- Find more than just crashes

What we would need

- Automatic generation of test cases ✓
- Filtering out invalid method calls ?
- Running test cases on the fly ✓
- Persisting test cases for later regression testing ✓
- Filtering out redundant test cases ✓
- Maximize combined coverage ✓
- Find more than just crashes ?

Fuzzing fits most of the criteria

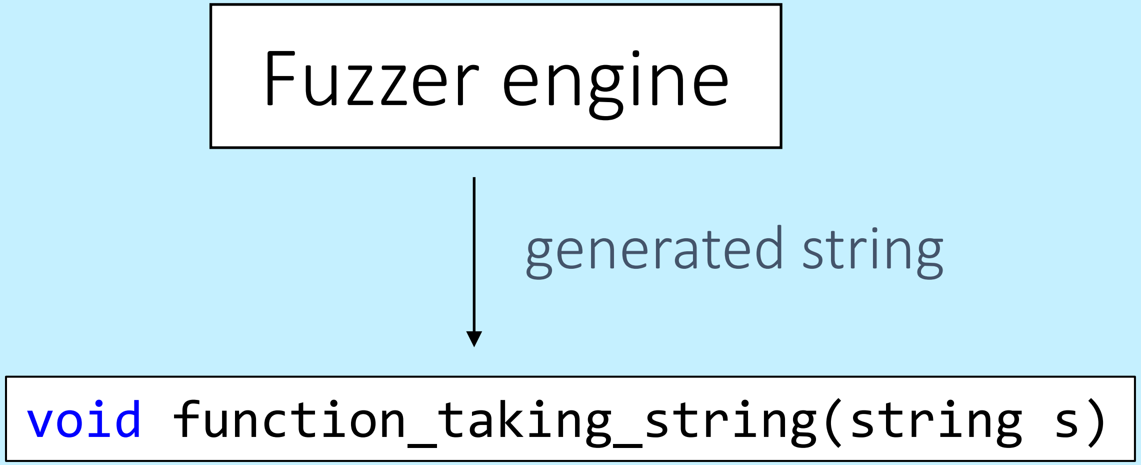
What is this "fuzzing"?

Additional info: "CppCon 2017: Kostya Serebryany \"Fuzz or lose...\""

Fuzzing in general

Fuzzer engine

Fuzzing in general

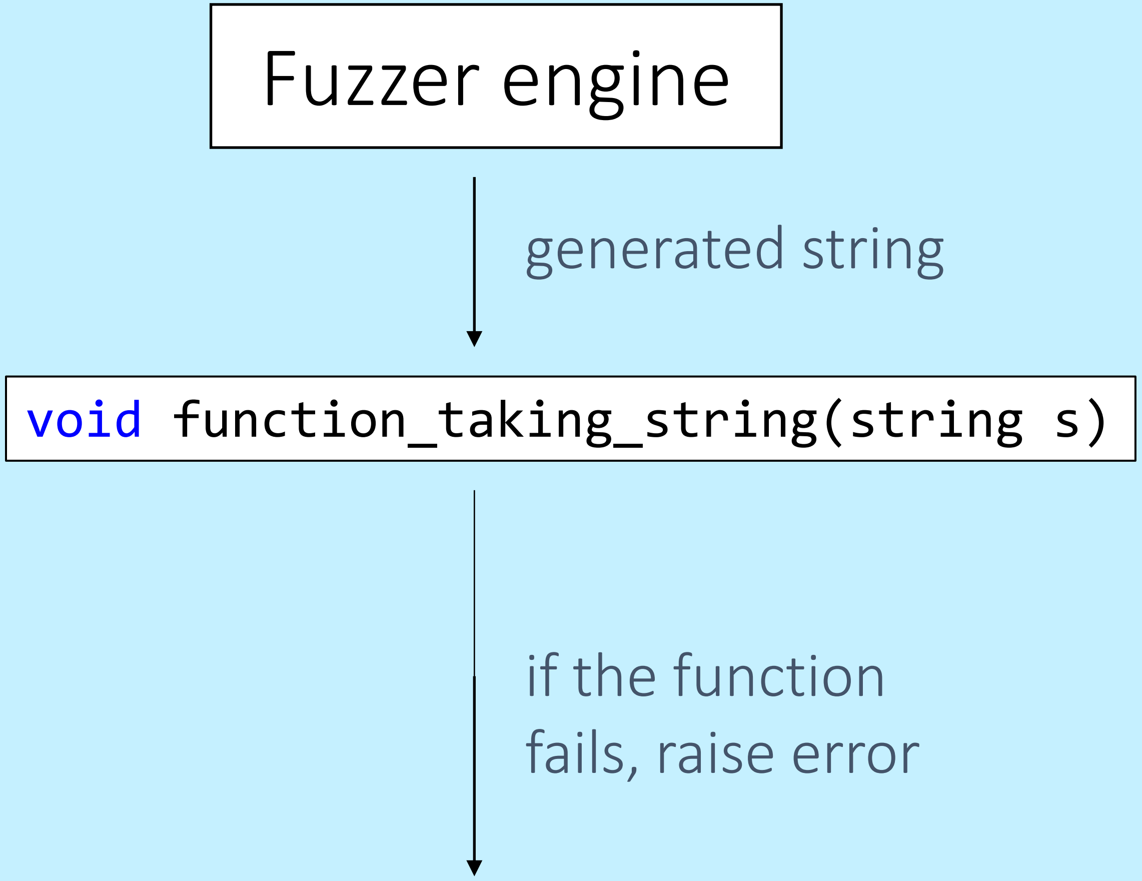


Fuzzer engine

generated string

```
void function_taking_string(string s)
```

Fuzzing in general



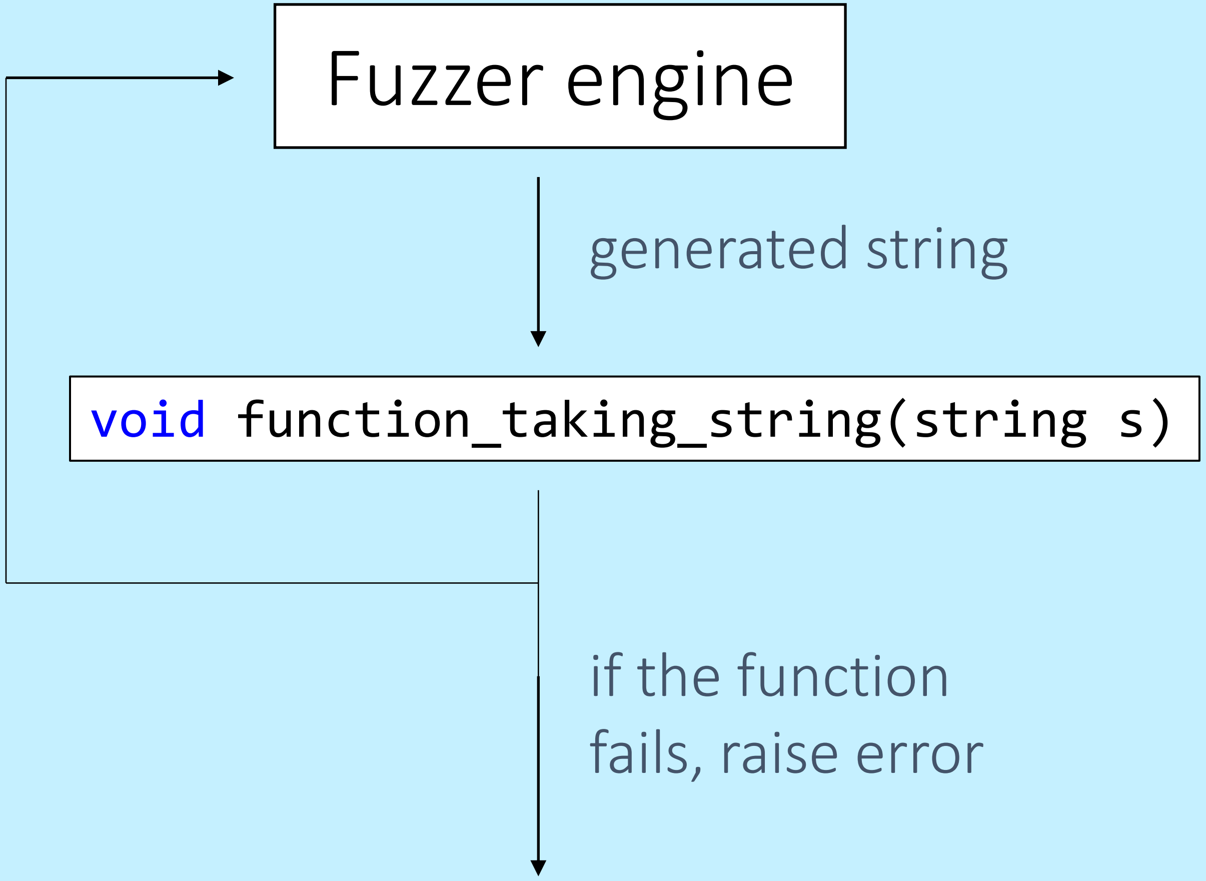
Fuzzer engine

generated string

```
void function_taking_string(string s)
```

if the function
fails, raise error

Fuzzing in general



Fuzzer engine

generated string

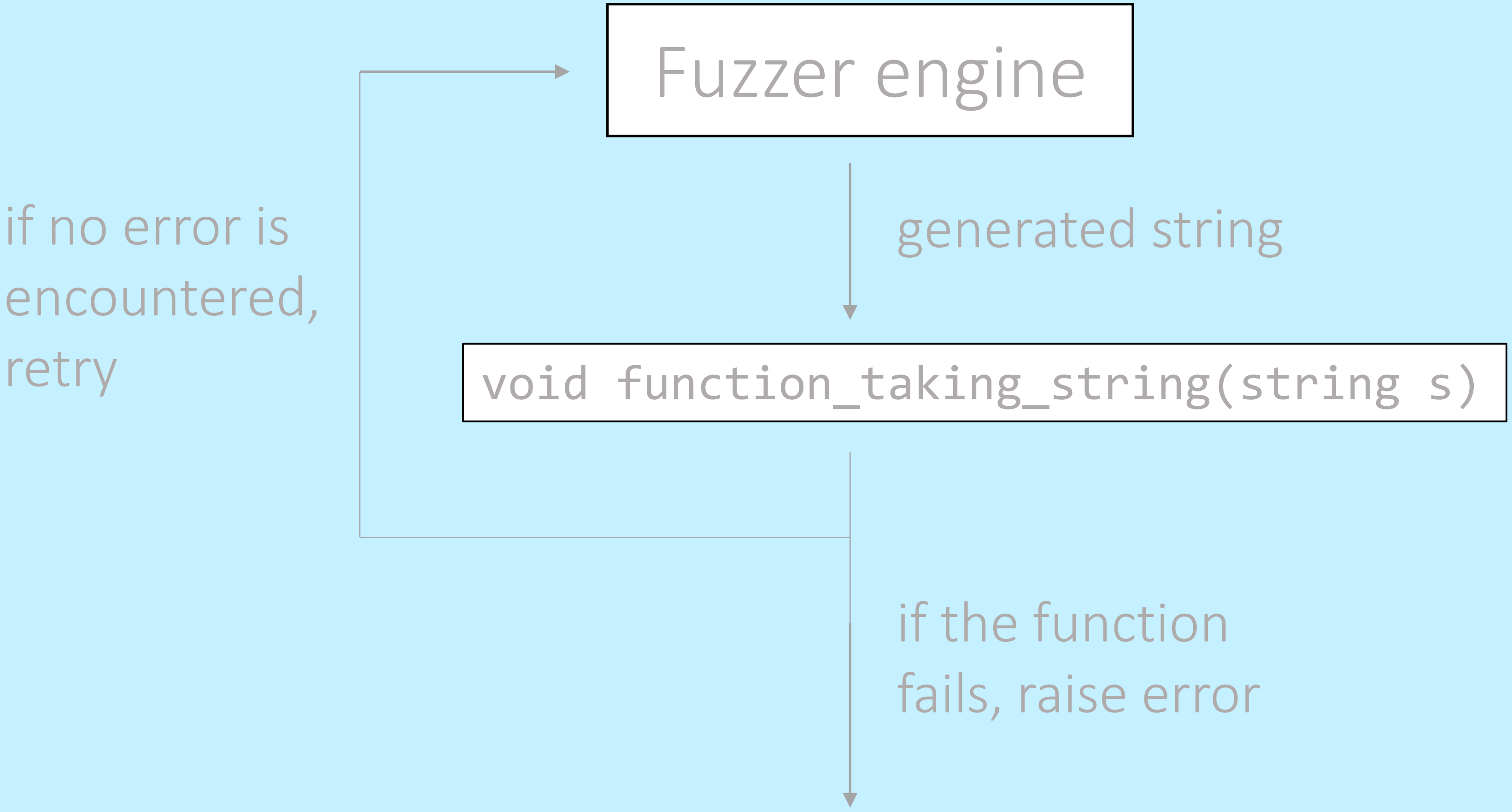
```
void function_taking_string(string s)
```

if no error is encountered, retry

if the function fails, raise error

Fuzzing methods have a proven track record, with thousands of crashes found

LLVM/Clang libFuzzer



Fuzzer engine

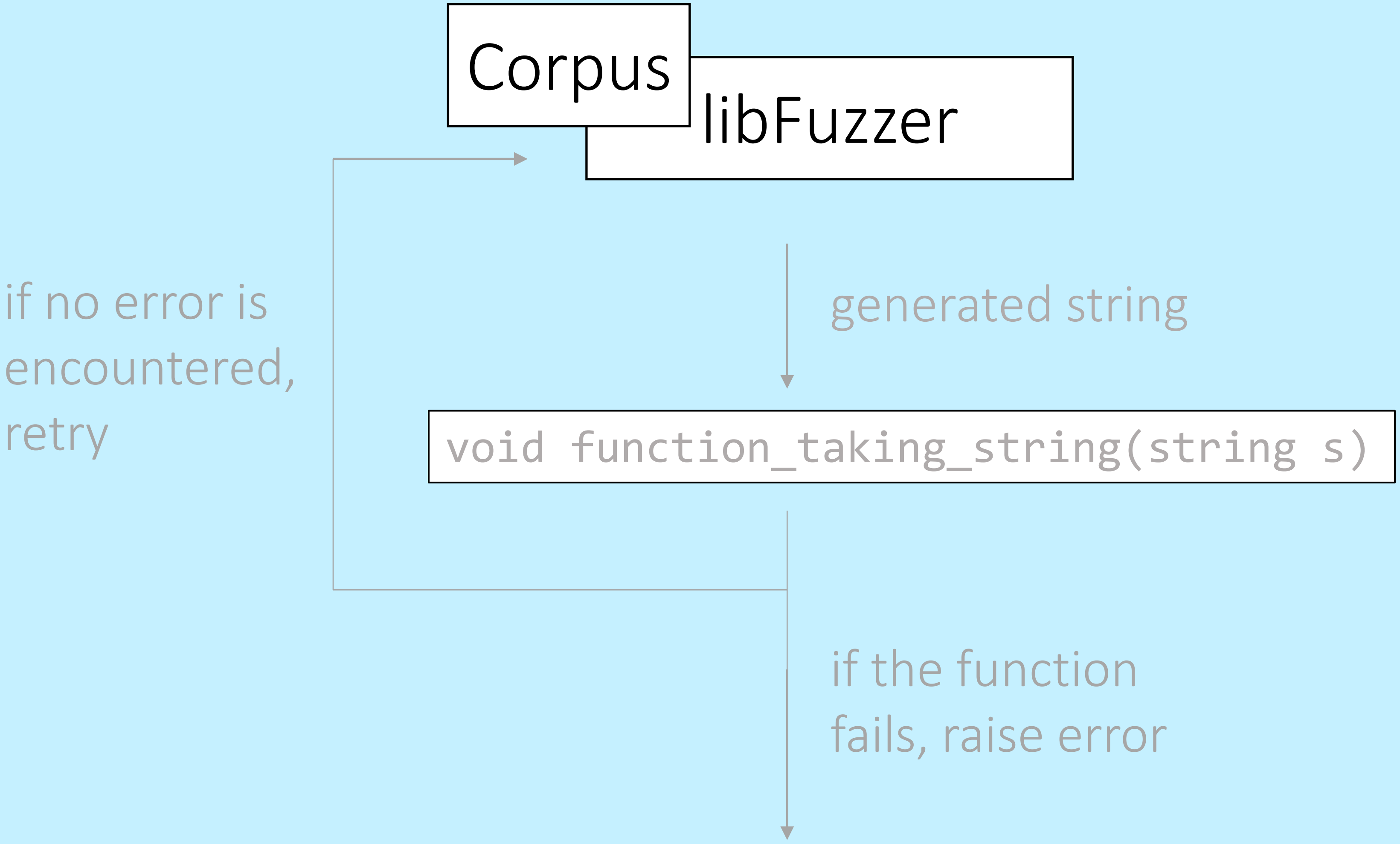
generated string

```
void function_taking_string(string s)
```

if no error is encountered, retry

if the function fails, raise error

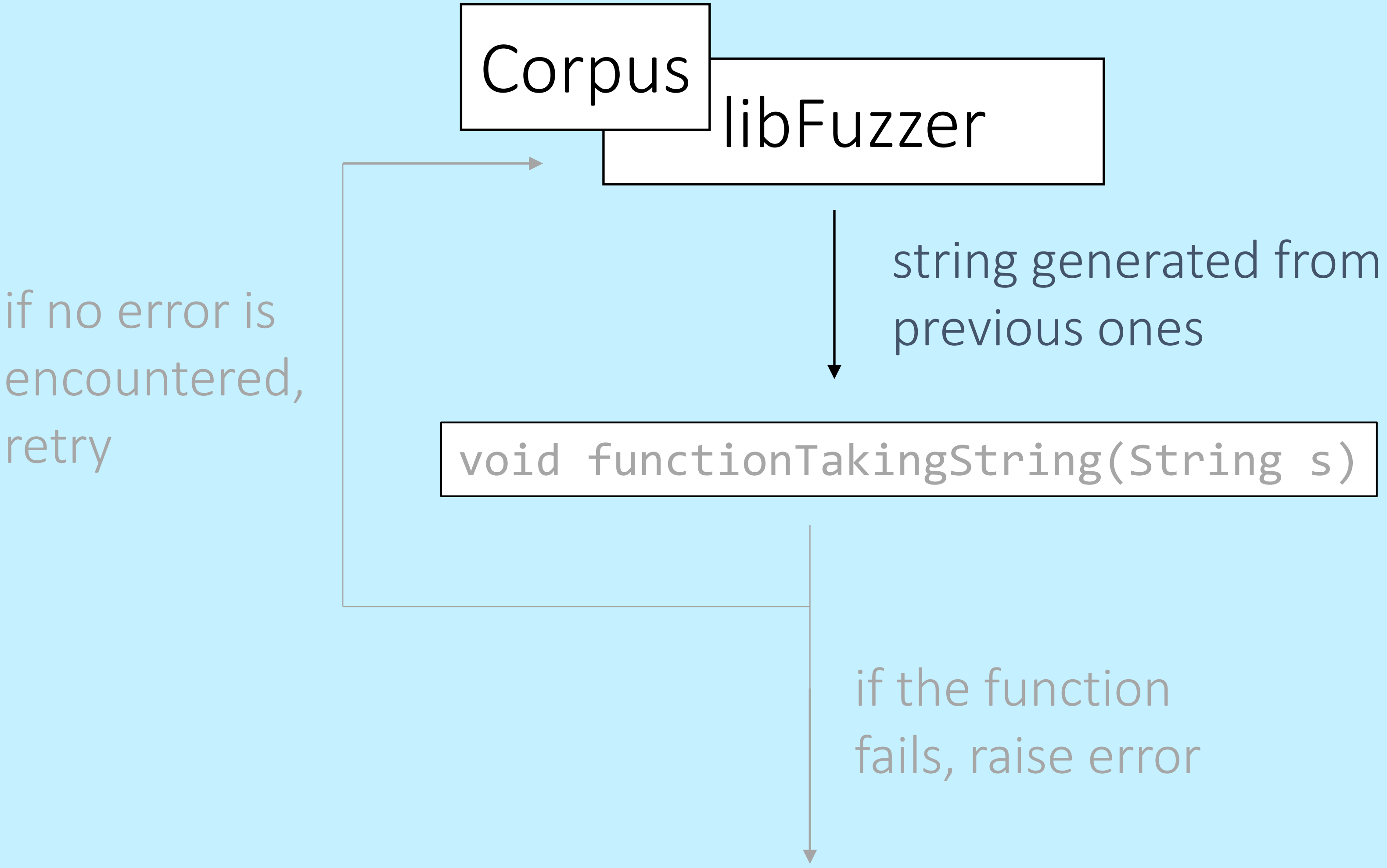
LLVM/Clang libFuzzer


Corpus
libFuzzer
generated string
if no error is encountered, retry
if the function fails, raise error

```
void function_taking_string(string s)
```

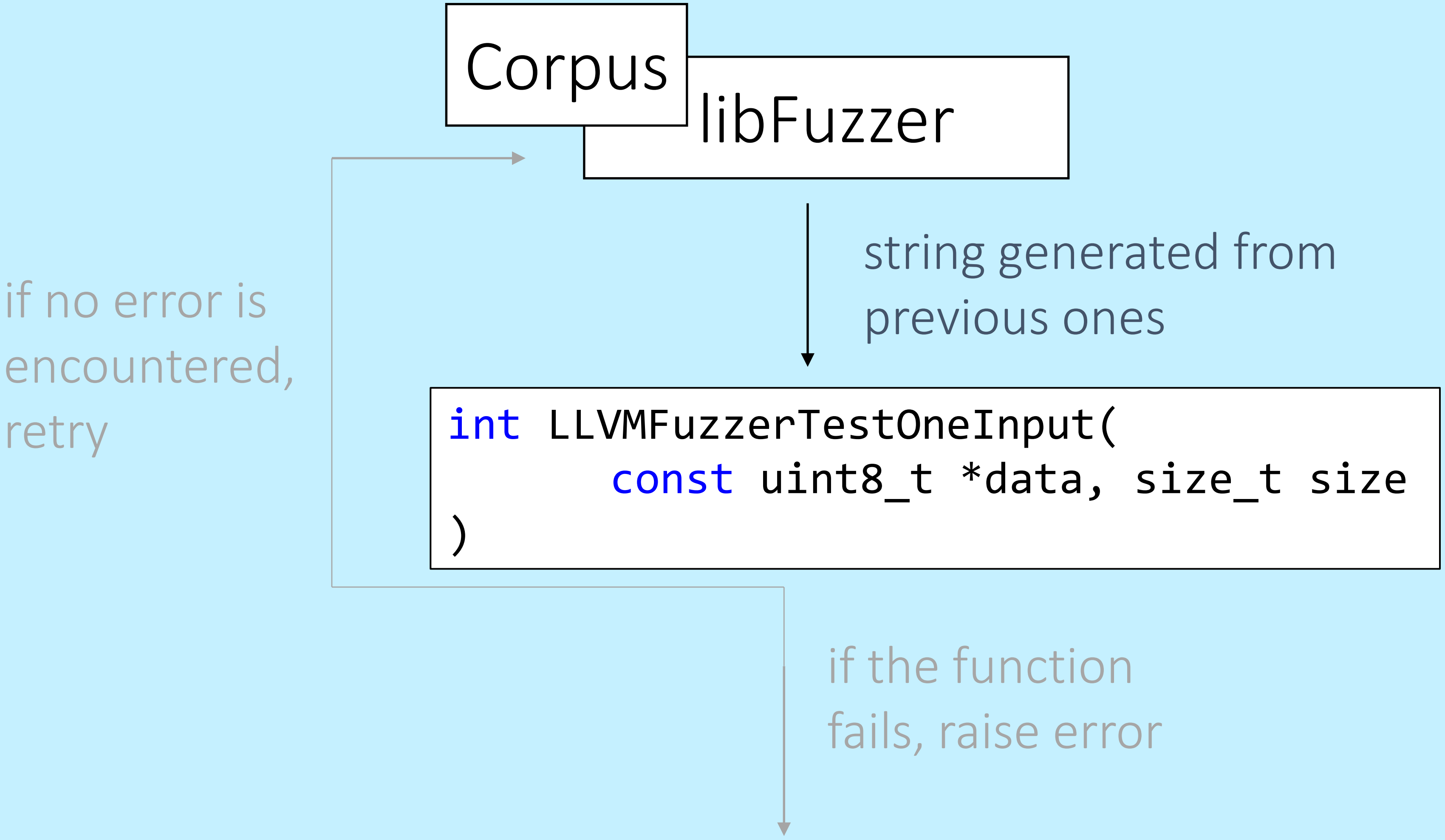
- maintains a set of interesting string arguments

LLVM/Clang libFuzzer


Corpus
libFuzzer
string generated from previous ones
void functionTakingString(String s)
if no error is encountered, retry
if the function fails, raise error

- maintains a set of string arguments which achieves maximum coverage
- uses coverage to guide the string generation

LLVM/Clang libFuzzer



Corpus

libFuzzer

string generated from previous ones

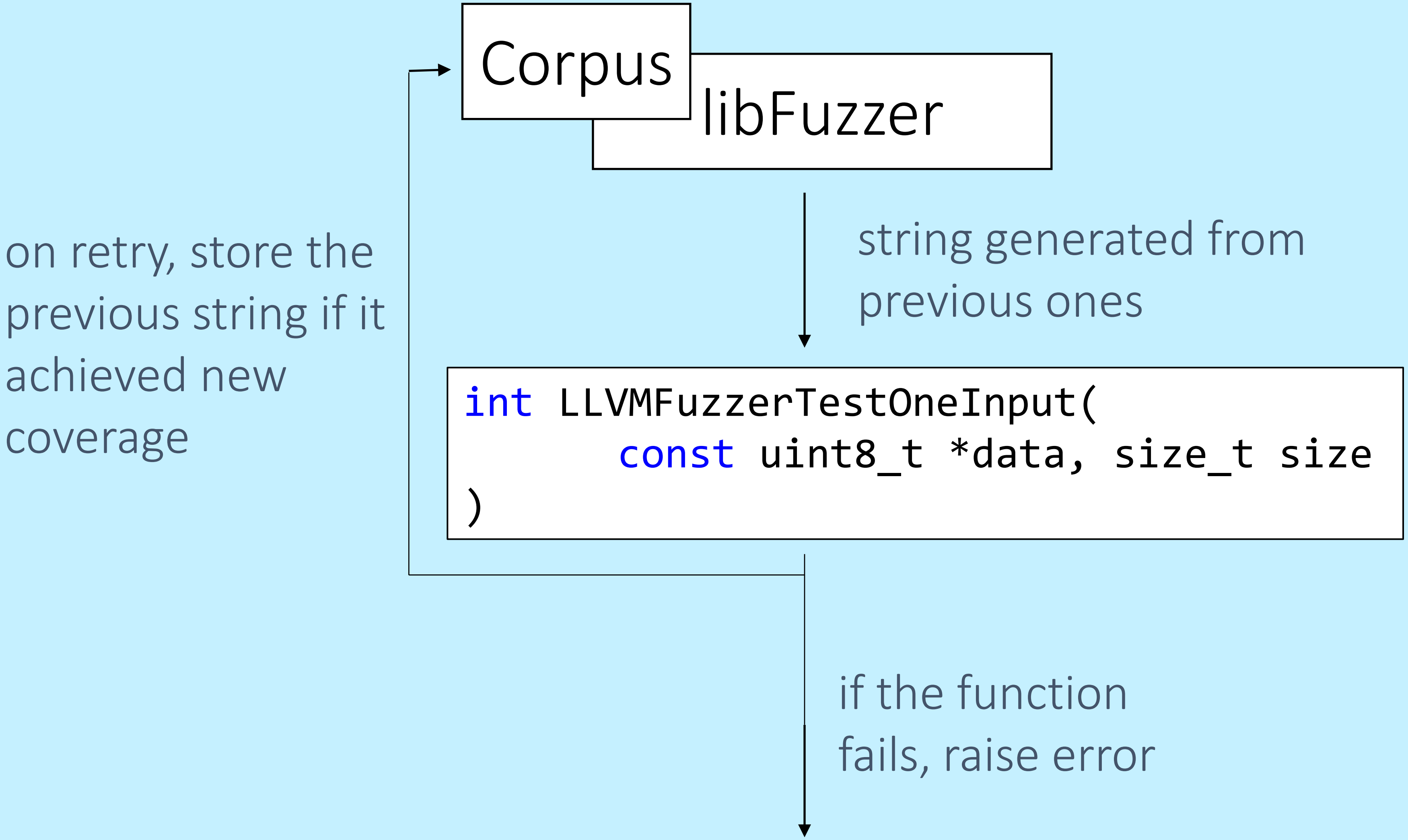
```
int LLVMFuzzerTestOneInput(
        const uint8_t *data, size_t size
)
```

if no error is encountered, retry

if the function fails, raise error

- maintains a set of string arguments which achieves maximum coverage
- uses coverage to guide the string generation

LLVM/Clang libFuzzer



Corpus

libFuzzer

string generated from previous ones

```
int LLVMFuzzerTestOneInput(
        const uint8_t *data, size_t size
)
```

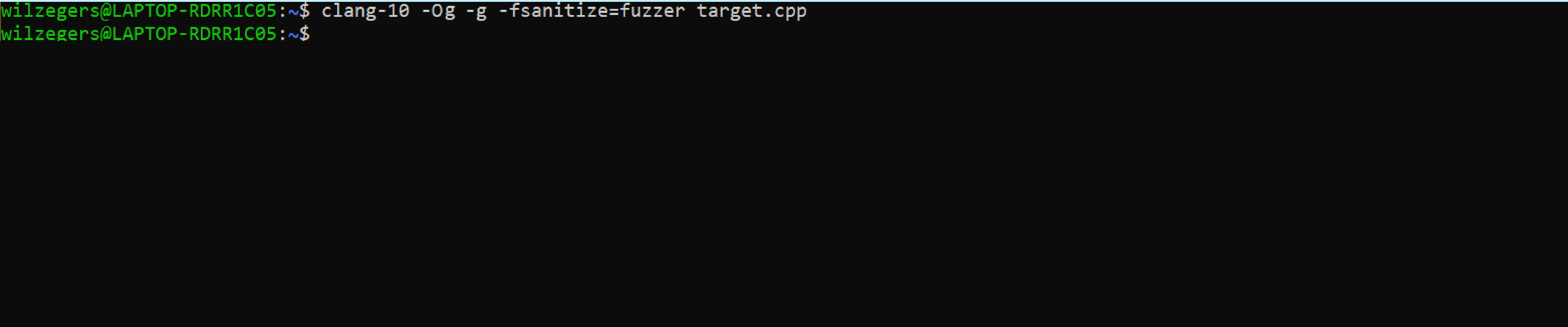
on retry, store the previous string if it achieved new coverage

if the function fails, raise error

- maintains a set of string arguments which achieves maximum coverage
- uses coverage to guide the string generation

Fuzzing example

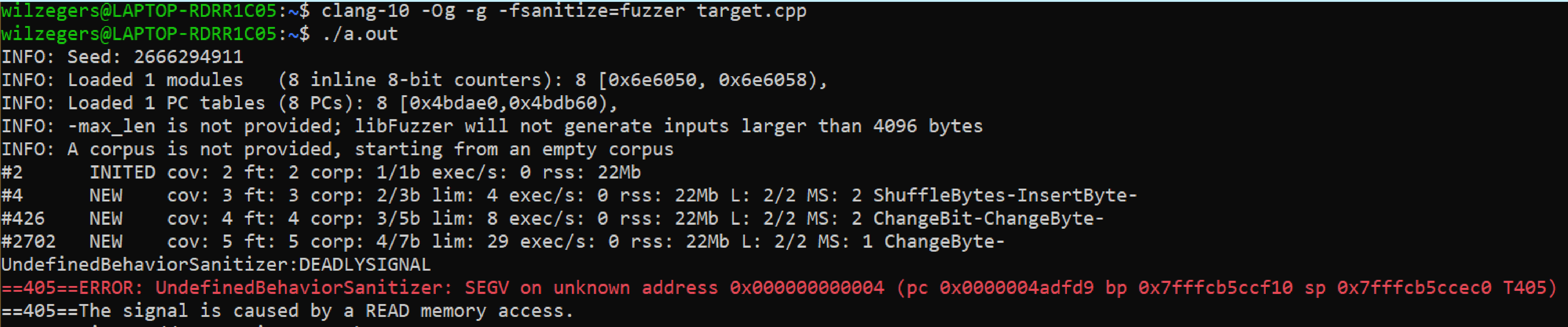
```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size > 1 && data[0] == '1' && data[1] == '2') {
        if (data[2] == '3') {
            static_cast<char*>(0)[4] = '4';
        }
    }
}
```



```
wilzegers@LAPTOP-RDRR1C05:~$ clang-10 -Og -g -fsanitize=fuzzer target.cpp
wilzegers@LAPTOP-RDRR1C05:~$
```

Fuzzing example

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size > 1 && data[0] == '1' && data[1] == '2') {
        if (data[2] == '3') {
            static_cast<char*>(0)[4] = '4';
        }
    }
}
```



```
wilzegers@LAPTOP-RDRR1C05:~$ clang-10 -Og -g -fsanitize=fuzzer target.cpp
wilzegers@LAPTOP-RDRR1C05:~$ ./a.out
INFO: Seed: 2666294911
INFO: Loaded 1 modules   (8 inline 8-bit counters): 8 [0x6e6050, 0x6e6058),
INFO: Loaded 1 PC tables (8 PCs): 8 [0x4bdae0,0x4bdb60),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 22Mb
#4      NEW    cov: 3 ft: 3 corp: 2/3b lim: 4 exec/s: 0 rss: 22Mb L: 2/2 MS: 2 ShuffleBytes-InsertByte-
#426    NEW    cov: 4 ft: 4 corp: 3/5b lim: 8 exec/s: 0 rss: 22Mb L: 2/2 MS: 2 ChangeBit-ChangeByte-
#2702   NEW    cov: 5 ft: 5 corp: 4/7b lim: 29 exec/s: 0 rss: 22Mb L: 2/2 MS: 1 ChangeByte-
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==405==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x000000000004 (pc 0x0000004adfd9 bp 0x7fffcb5ccf10 sp 0x7fffcb5ccec0 T405)
==405==The signal is caused by a READ memory access.
```

Clang Sanitizers

Sanitizers are compiler build-in error detectors with relatively small runtime cost. Clang has

- AddressSanitizer - use-after-free, double-free, ...
- MemorySanitizer - uninitialized reads
- UndefinedBehaviourSanitizer - overflows, divide by zero, ...
- ThreadSanitizer - data races

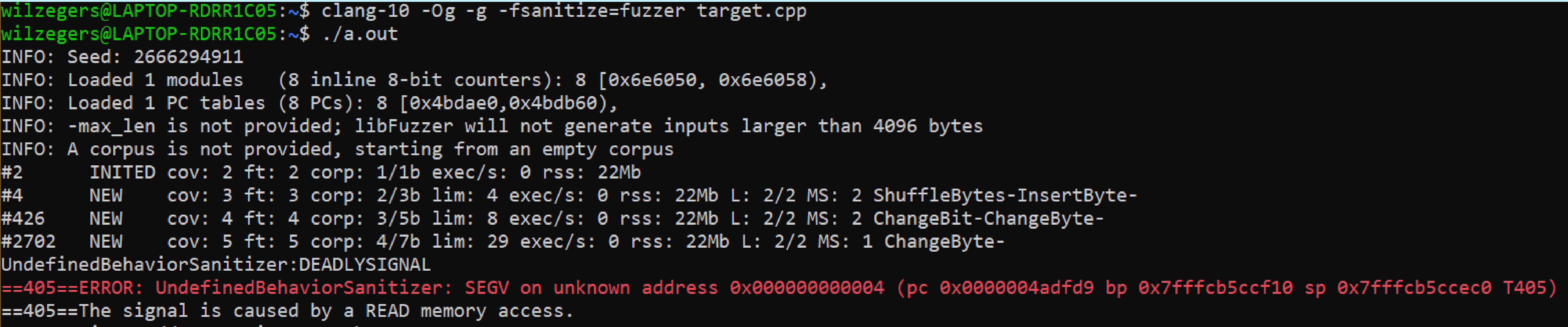
Turning them on:

```
$ clang -g -fsanitize=fuzzer,memory target.cpp
```

Additional info: "CppCon 2014: Kostya Serebryany \"Sanitize your C++ code\""

Fuzzing example - UBSanitizer

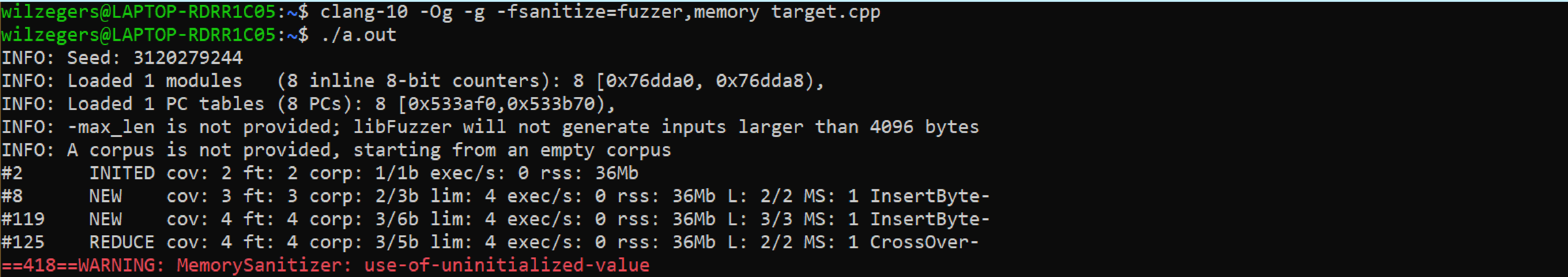
```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size > 1 && data[0] == '1' && data[1] == '2') {
        if (data[2] == '3') {
            static_cast<char*>(0)[4] = '4';
        }
    }
}
```



```
wilzegers@LAPTOP-RDRR1C05:~$ clang-10 -Og -g -fsanitize=fuzzer target.cpp
wilzegers@LAPTOP-RDRR1C05:~$ ./a.out
INFO: Seed: 2666294911
INFO: Loaded 1 modules   (8 inline 8-bit counters): 8 [0x6e6050, 0x6e6058),
INFO: Loaded 1 PC tables (8 PCs): 8 [0x4bdae0,0x4bdb60),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 22Mb
#4      NEW    cov: 3 ft: 3 corp: 2/3b lim: 4 exec/s: 0 rss: 22Mb L: 2/2 MS: 2 ShuffleBytes-InsertByte-
#426    NEW    cov: 4 ft: 4 corp: 3/5b lim: 8 exec/s: 0 rss: 22Mb L: 2/2 MS: 2 ChangeBit-ChangeByte-
#2702   NEW    cov: 5 ft: 5 corp: 4/7b lim: 29 exec/s: 0 rss: 22Mb L: 2/2 MS: 1 ChangeByte-
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==405==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x000000000004 (pc 0x0000004adfd9 bp 0x7fffcb5ccf10 sp 0x7fffcb5ccec0 T405)
==405==The signal is caused by a READ memory access.
```

Fuzzing example - MemorySanitizer

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size > 1 && data[0] == '1' && data[1] == '2') {
        if (data[2] == '3') {
            static_cast<char*>(0)[4] = '4';
        }
    }
}
```



```
wilzegers@LAPTOP-RDRR1C05:~$ clang-10 -Og -g -fsanitize=fuzzer,memory target.cpp
wilzegers@LAPTOP-RDRR1C05:~$ ./a.out
INFO: Seed: 3120279244
INFO: Loaded 1 modules   (8 inline 8-bit counters): 8 [0x76dda0, 0x76dda8),
INFO: Loaded 1 PC tables (8 PCs): 8 [0x533af0,0x533b70),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 36Mb
#8      NEW    cov: 3 ft: 3 corp: 2/3b lim: 4 exec/s: 0 rss: 36Mb L: 2/2 MS: 1 InsertByte-
#119    NEW    cov: 4 ft: 4 corp: 3/6b lim: 4 exec/s: 0 rss: 36Mb L: 3/3 MS: 1 InsertByte-
#125    REDUCE cov: 4 ft: 4 corp: 3/5b lim: 4 exec/s: 0 rss: 36Mb L: 2/2 MS: 1 CrossOver-
==418==WARNING: MemorySanitizer: use-of-uninitialized-value
```

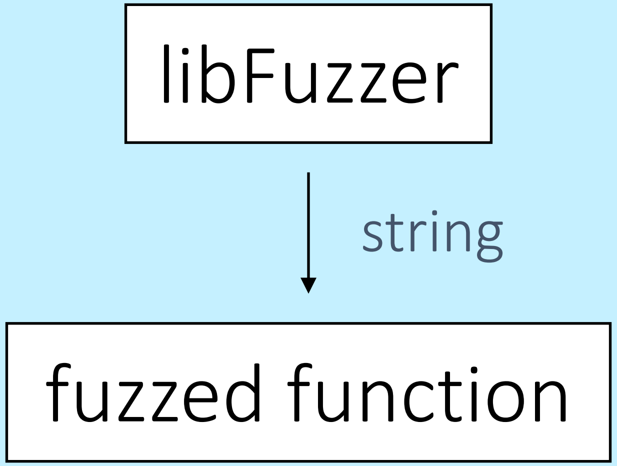
What we would need

- Automatic generation of test cases ✓
- Filtering out invalid method calls ?
- Running test cases on the fly ✓
- Persisting test cases for later regression testing ✓
- Filtering out redundant test cases ✓
- Maximize combined coverage ✓
- Find more than just crashes ?

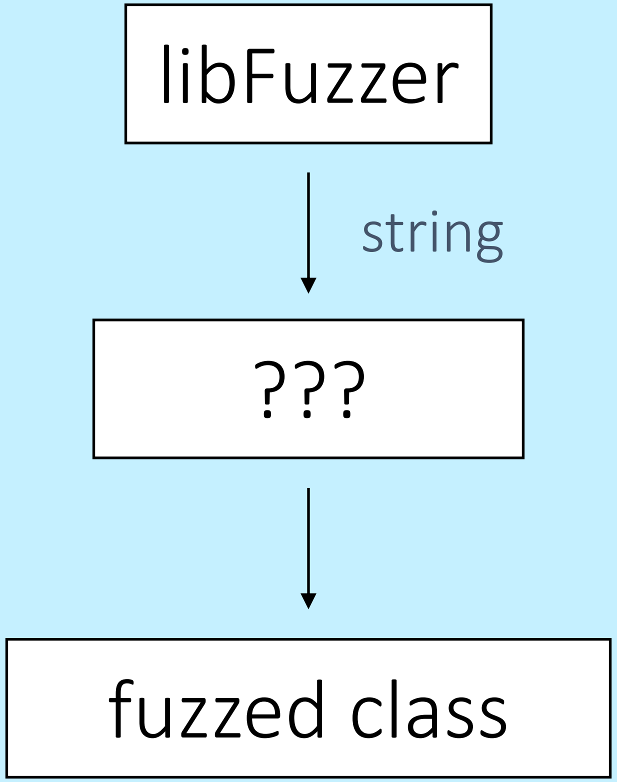
Fuzzing fits most of the criteria

How does fuzzing help us?
We won't be testing just
string interfaces...

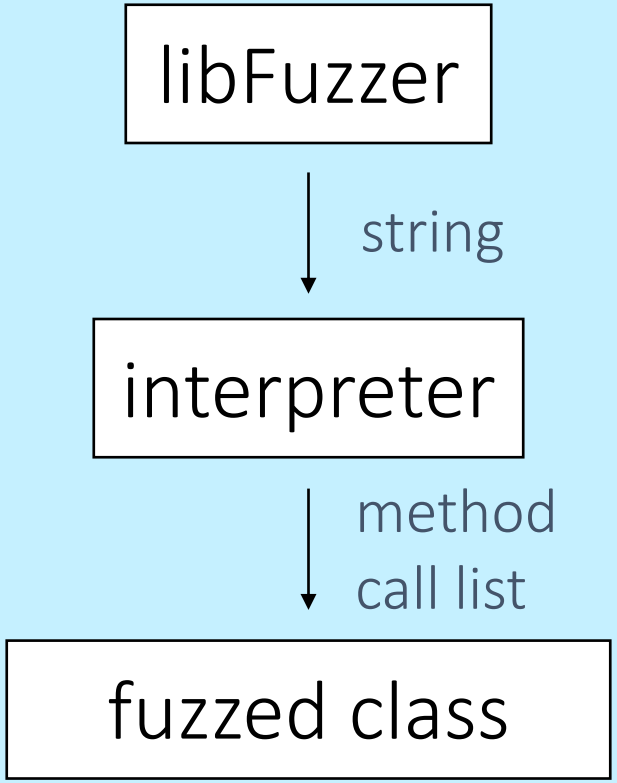
Transforming libFuzzer to an Interface Fuzzer



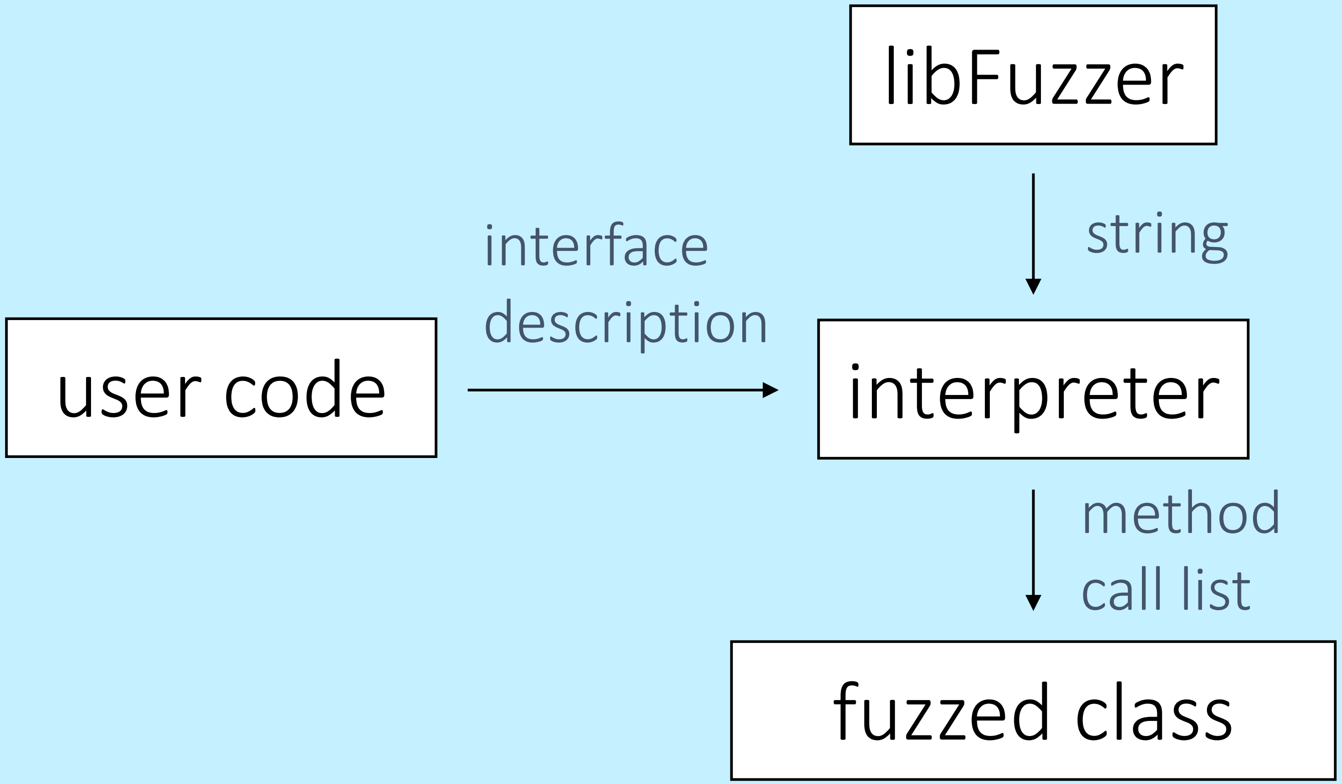
Transforming libFuzzer to an Interface Fuzzer



Transforming libFuzzer to an Interface Fuzzer



Transforming libFuzzer to an Interface Fuzzer



Interface Description

List of the methods

```
Autotest::Builder<my_deque>
    .CONST_FUN(size)
    .CONST_FUN(back)
    .CONST_FUN(front)
    .FUN(pop_back)
    .FUN(pop_front)
    .FUN(push_back)
    .FUN(push_front)
```

Interface Description

List of the methods, their preconditions

```
auto not_empty = [](const auto& self) {
    return self.size() > 0;
};

Autotest::Builder<my_deque>
    .CONST_FUN(size)
    .CONST_FUN(back).If(not_empty)
    .CONST_FUN(front).If(not_empty)
    .FUN(pop_back).If(not_empty)
    .FUN(pop_front).If(not_empty)
    .FUN(push_back)
    .FUN(push_front)
```

Interface Description

List of the methods, their preconditions and "argument placeholders"

```
auto not_empty = [](const auto& self) {
    return self.size() > 0;
};

Autotest::Builder<my_deque>
    .CONST_FUN(size)
    .CONST_FUN(back).If(not_empty)
    .CONST_FUN(front).If(not_empty)
    .FUN(pop_back).If(not_empty)
    .FUN(pop_front).If(not_empty)
    .FUN(push_back, integral<int>)
    .FUN(push_front, integral<int>)
```

Interface Description

List of the methods, their preconditions and "argument placeholders"

```
auto not_empty = [](const auto& self) {
    return self.size() > 0;
};

Autotest::Builder<my_deque>
    .CONST_FUN(size)
    .CONST_FUN(back).If(not_empty)
    .CONST_FUN(front).If(not_empty)
    .FUN(pop_back).If(not_empty)
    .FUN(pop_front).If(not_empty)
    .FUN(push_back, integral<int>)
    .FUN(push_front, integral<int>)
```

0	`std::size_t size() const`
1	`int back() const`
2	`int front() const`
3	`void pop_back()`
4	`void pop_front()`
5	`void push_back(int)`
6	`void push_front(int)`

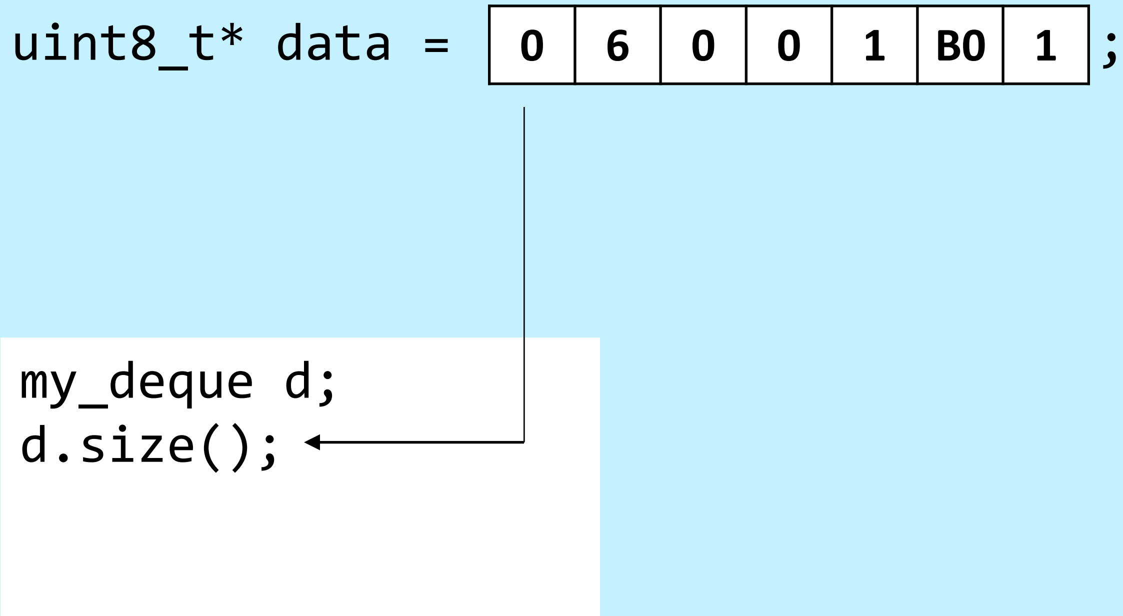
Interpreting the Generated Strings

```
uint8_t* data = | 0 | 6 | 0 | 0 | 1 | B0 | 1 | ;
```

Process:

- Choose nth method with a satisfied precondition

Interpreting the Generated Strings



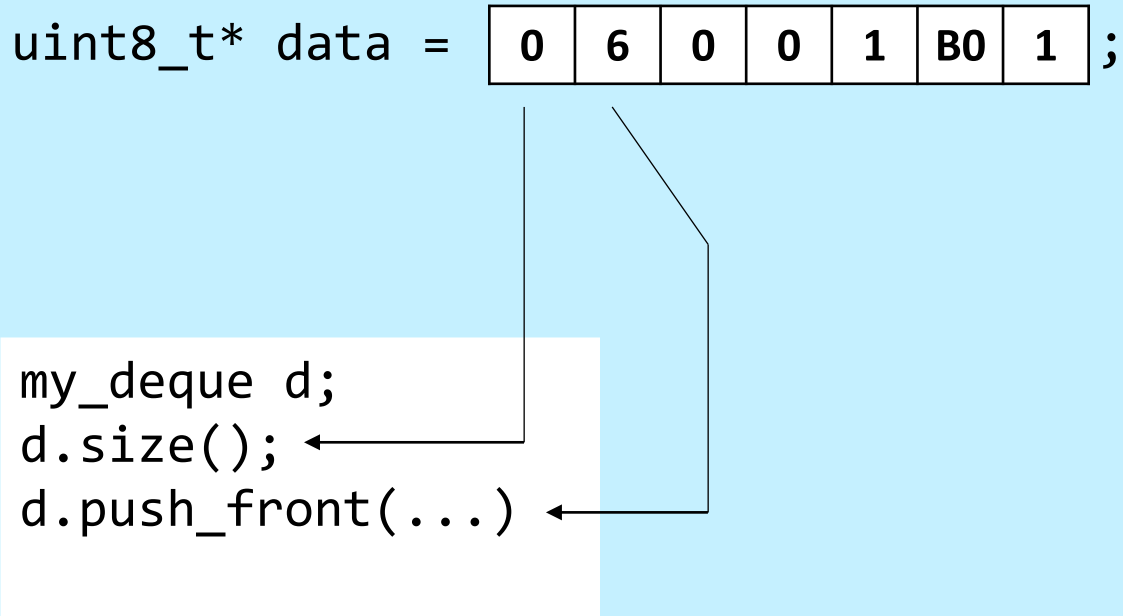
```
uint8_t* data = | 0 | 6 | 0 | 0 | 1 | B0 | 1 | ;
```

```
my_deque d;
d.size();
```

Process:

- Choose nth method with a satisfied precondition

Interpreting the Generated Strings



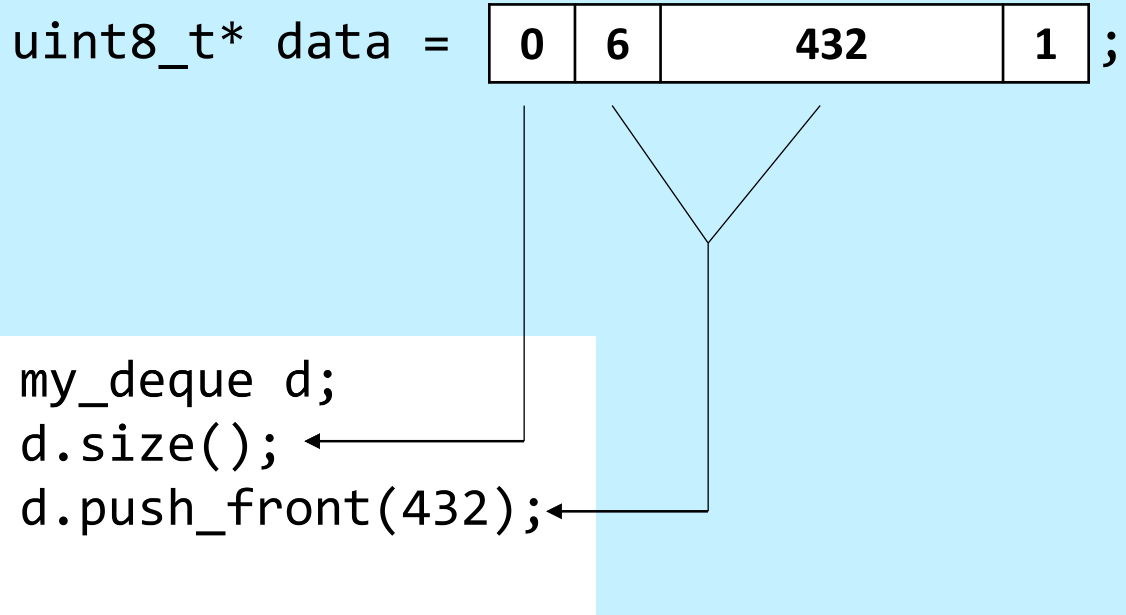
```
uint8_t* data = | 0 | 6 | 0 | 0 | 1 | B0 | 1 | ;
```

```
my_deque d;
d.size();
d.push_front(...)
```

Process:

- Choose n^th method with a satisfied precondition

Interpreting the Generated Strings



uint8_t* data =

0	6	432	1

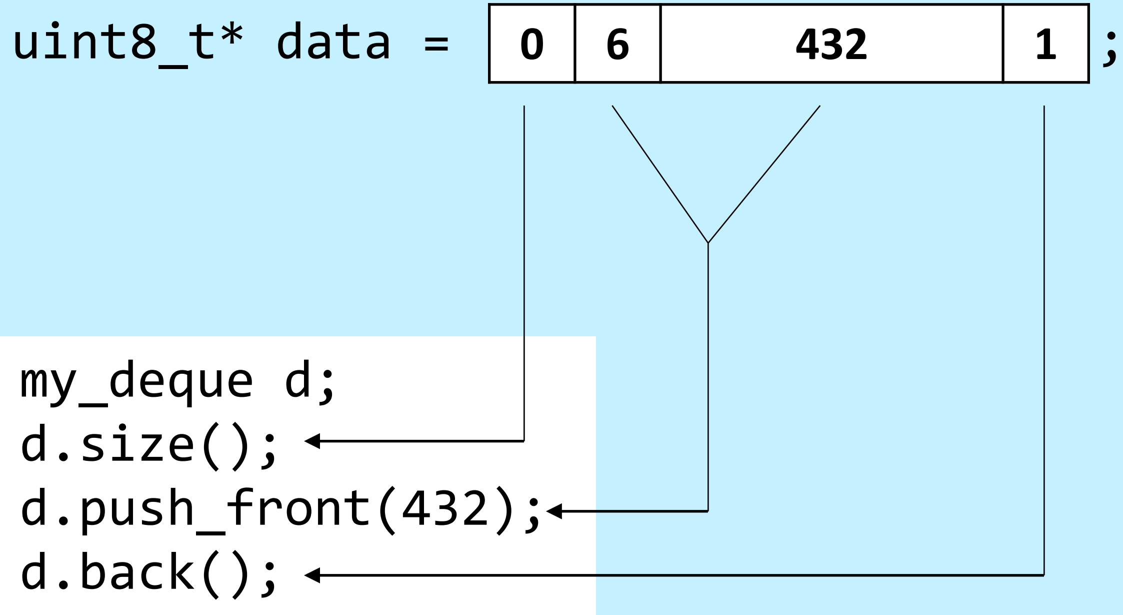
;

```
my_deque d;
d.size();
d.push_front(432);
```

Process:

- Choose nth method with a satisfied precondition
- Read arguments from the generated data

Interpreting the Generated Strings



```
uint8_t* data = 
```

0	6	432	1

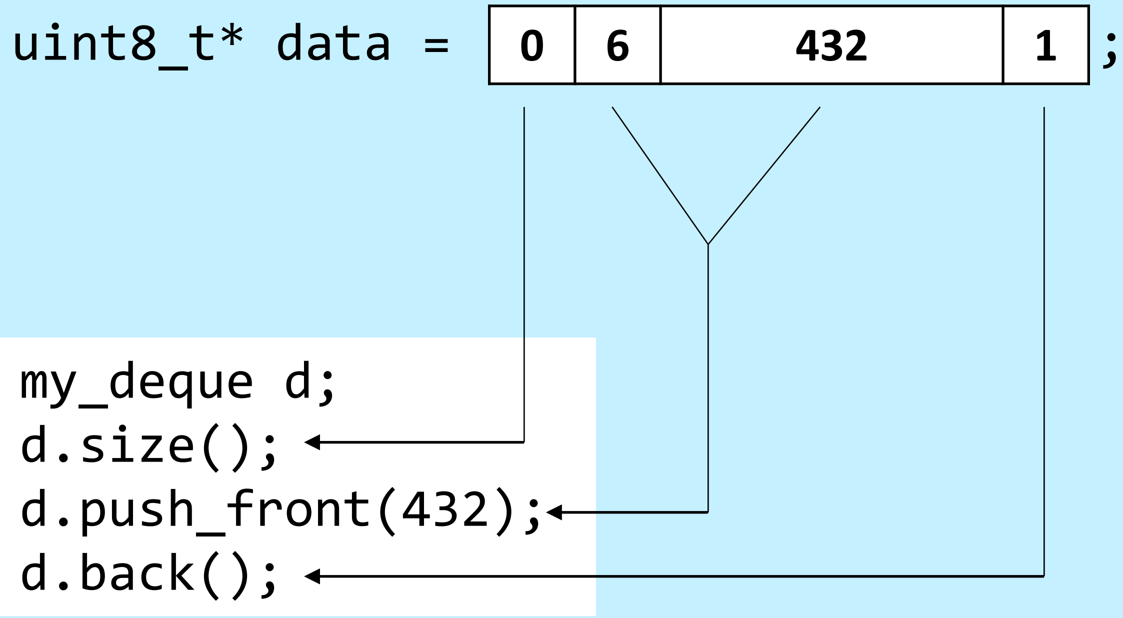
```
;
```

```
my_deque d;
d.size();
d.push_front(432);
d.back();
```

Process:

- Choose nth method with a satisfied precondition
- Read arguments from the generated data

Interpreting the Generated Strings



```
my_deque d;
d.size();
d.push_front(432);
d.back();
```

Process:

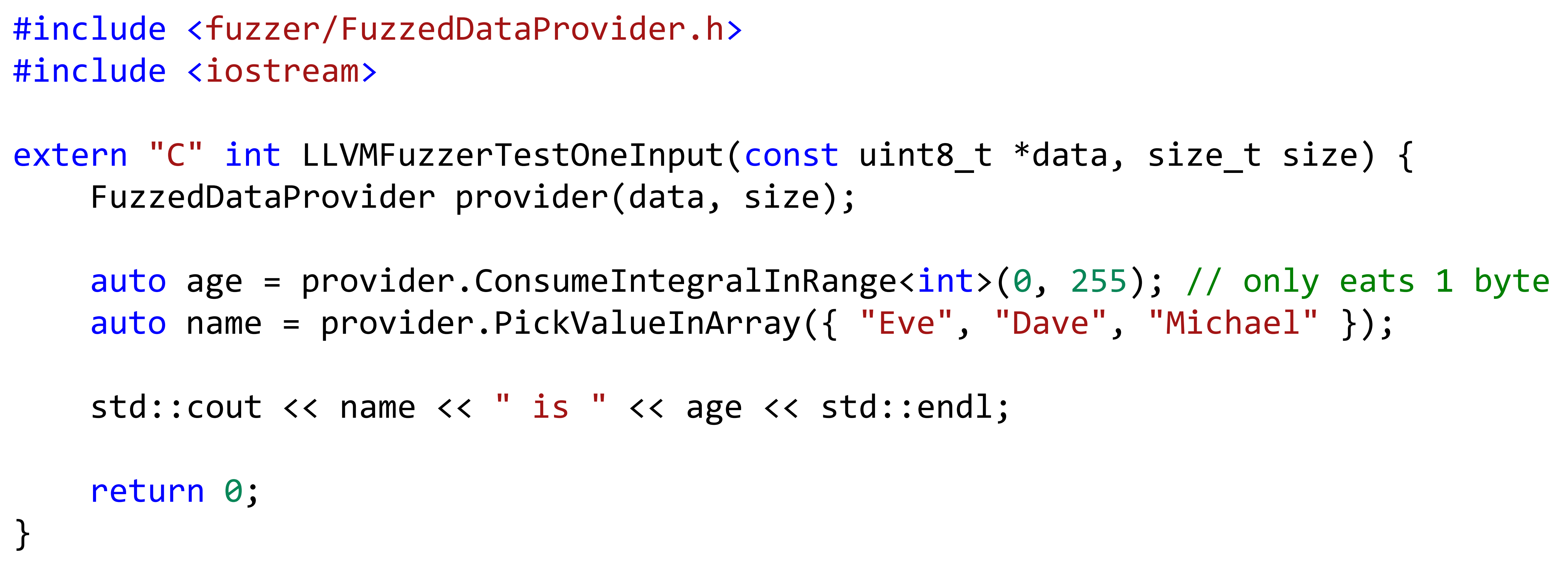
- Choose n^{th} method with a satisfied precondition
- Read arguments from the generated data

Edge cases:

- Integer too big
- Not enough bytes available

libFuzzer utilities to the rescue!

libFuzzers FuzzedDataProvider



```cpp
#include <fuzzer/FuzzedDataProvider.h>
#include <iostream>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    FuzzedDataProvider provider(data, size);

    auto age = provider.ConsumeIntegralInRange<int>(0, 255); // only eats 1 byte
    auto name = provider.PickValueInArray({ "Eve", "Dave", "Michael" });

    std::cout << name << " is " << age << std::endl;

    return 0;
}
```

The argument placeholders are implemented as T(FuzzedDataProvider&) functions

Crashes are not Enough

Currently the tool is generating test cases like the following

```
MyDeque md;

md.push_back(123)m;
md.push_back(-76);
md.push_back(0);
md.pop_back();
```

```
MyDeque md;

md.push_front(456);
md.front();
md.back();
```

```
MyDeque md;

md.push_back(1);
md.push_front(2);
md.push_back(3);
md.push_front(4);
md.pop_back();
md.pop_front();
md.push_back(5);
md.push_front(6);
```

What if we have bugs which are not revealed by crashes?

Invariants

An invariant is a condition that is always true as long as the object is in a valid state (e.g. in the deque the front position must be <= the back)

```
TEST(my_deque_test, push_pop)
{
    my_deque md;
    ASSERT_EQ(md.size(), 0);

    md.push_back(42);
    ASSERT_EQ(md.size(), 1);
    ASSERT_EQ(md.back(), 42);

    md.pop_back();
    ASSERT_EQ(md.size(), 0);
}
```

Invariants

An invariant is a condition that is always true as long as the object is in a valid state (e.g. in the deque the front position must be <= the back)

```
TEST(my_deque_test, push_pop)
{
    my_deque md;
    ASSERT_EQ(md.size(), 0);

    md.push_back(42);
    ASSERT_EQ(md.size(), 1);
    ASSERT_EQ(md.back(), 42);

    md.pop_back();
    ASSERT_EQ(md.size(), 0);
}
```

```
MyDeque md;

checkInvariant(md);
md.push_front(456);
checkInvariant(md);
md.front();
checkInvariant(md);
md.back();
checkInvariant(md);
```

Invariants

An invariant is a condition that is always true as long as the object is in a valid state (e.g. in the deque the front position must be <= the back)

```
TEST(my_deque_test, push_pop)
{
    my_deque md;
    ASSERT_EQ(md.size(), 0);

                back(), 42);

    md.pop_back();
    ASSERT_EQ(md.size(), 0);
}
```

```
 MyDeque d;

     Invariant(d);
 d.front();
 checkInvariant(d);
 d.back();
 checkInvariant(d);
```

Of course, this is not as strong of a check as manual assertions in unit tests.

Let's put it to the test

Case Study 1: Simplified Deque

- Single-ended "deque": A single ended queue implemented with a vector of static arrays. Shares problems with a real deque.
  - 100% code coverage reached reliably within seconds

```
template<class T>
struct block_array {

    template<class... Args>
    void emplace_back(Args&&... args);
    void pop_back();
    T& back();
    const T& back() const;

   std::size_t size() const;
private:
    // ...
};
```

Case Study 1: Simplified Deque

```
#include "autotest/autotest.hpp"
#include "block-array.hpp"

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    auto not_empty = [](const auto& self) {
        return self.size() > 0;
    };
    AutoTest::Builder<block_array<int>>{ data, size }
        .AUTOTEST_FUN(emplace_back, AutoTest::Args::integral<int>)
        .AUTOTEST_FUN(pop_back).If(not_empty)
        .AUTOTEST_FUN(back).If(not_empty)
        .AUTOTEST_CONST_FUN(back).If(not_empty)
        .AUTOTEST_CONST_FUN(size)
    .execute();
    return 0;
}
```

Case Study 2: Robin-hood Hash Map

- A Robin-hood hash map implementation: A state-of-the-art hash map implementation, according to some measurements one of the fastest currently available (header only, +2200 lines).
  - fluctuating 88%-93% code coverage reached within a minute
  - after fine-tuning fuzzing parameters, 93% code-coverage reached reliably within a minute
  - the remaining 7% did not seem reachable in the test

hash-map library source: https://github.com/martinus/robin-hood-hashing

Case Study 2: Robin-hood Hash Map

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    using hash_map = robin_hood::unordered_flat_map<
        std::string, std::string
    >;
    auto to_res = Autotest::Args::integralRange(1, 10000);
    auto key = Autotest::Args::randomString(20);
    auto key_val = [](auto& state) {
        return robin_hood::pair<std::string, std::string>(
            Autotest::Args::randomString(20)(state),
            Autotest::Args::randomString(20)(state)
        );
    };

    // ...

}
```

hash-map library source: https://github.com/martinus/robin-hood-hashing

Case Study 2: Robin-hood Hash Map

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {

    // ...

    Autotest::Builder<hash_map>{ data, size }
        .AUTOTEST_FUN(insert, key_val)
        .AUTOTEST_FUN(emplace, key, key)
        .AUTOTEST_CONST_FUN(count, key)
        .AUTOTEST_CONST_FUN(contains, key)
        .AUTOTEST_FUN(erase, key)
        .AUTOTEST_FUN(reserve, to_res)
        .AUTOTEST_FUN(rehash, to_res)
        .AUTOTEST_CONST_FUN(find, key)
    .execute();
    return 0;
}
```

hash-map library source: https://github.com/martinus/robin-hood-hashing

Summary

- Problem: no tools for mass-testing classes
- Solution: adapt existing fuzzing technology
  - create interface description
  - interpret fuzz string as method list
  - exclude invalid states based on precondition
  - execute interpreted test case
  - smooth interoperation with sanitizers
- First results seem promising
- prototype available at https://gitlab.com/wilzegers/autotest/

Future work

- What about non-containers?
- Make the prototype less prototype-y
- Answering any questions you may have

Future work

- What about non-containers?
- Make the prototype less prototype-y
- Answering any questions you may have

Thank you for your attention

Feel free to contact me (Barnabás Bágyi) at bagyibarna@gmail.com