DeepState: Bringing vulnerability detection tools into the development lifecycle

Peter Goodman (Trail of Bits)
Gustavo Grieco (Trail of Bits)
Alex Groce (Northern Arizona University)

Introductions

Peter Goodman
Senior Security Engineer

peter@trailofbits.com
Trail of Bits

Gustavo Grieco
Security Engineer

gustavo.grieco@trailofbits.com
Trail of Bits

Alex Groce
Associate Professor

alex.groce@nau.edu
Northern Arizona University

Today's workshop is interactive (1)

Before beginning, please do one of the following in a terminal on your computers:

Clone the ieee_secdev_2018 branch:

```
git clone https://github.com/trailofbits/deepstate -b ieee_secdev_2018
```

OR

Download and extract:

https://github.com/trailofbits/deepstate/archive/ieee_secdev_2018.zip

Today's workshop is interactive (2)

Go into the cloned/unzipped `deepstate` directory, and execute the following:

```
$ vagrant up
$ vagrant ssh
```

If successful, this is what you should see:

```
vagrant@ubuntu-xenial $
```

How do developers test code?

- **Static Analysis**
  - Many tools available, most are commercial (e.g. Coverity)
  - False positives continue to be a vexing problem
  - 57% have never used one (JetBrains Survey)
- **Unit Tests!**
  - Tooling is free
  - Test for functionality and security
  - Nearly everyone is familiar with the concepts
  - Only 29% do not use unit tests (JetBrains Survey)

Unit testing is great!

- **Unit tests are a software assurance methodology**
  - Typically test individual functions, classes, or groups of related functionality
  - As code changes (e.g. improving an algorithm), unit tests help to ensure that expected functionality or results remain the same

Let's write a unit test (1)

Enter the exercises directory and open FirstTest.cpp

```
vagrant@ubuntu-xenial $ cd exercises
vagrant@ubuntu-xenial $ nano FirstTest.cpp
```

Let's write a unit test (2)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!
}
```

Let's write a unit test (3)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!
}
```

Function that we want to test

Let's write a unit test (4)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!
}
```

Test case

Let's write a unit test (5)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!
}
```

Test case name and test name

Let's write a unit test (6)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!
}
```

Assertions verifying output is as expected

Let's write a unit test (7)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!
}
```

Homework!!!

Executing your first test (1)

Please save and close FirstTest.cpp, and execute the following command:

```
vagrant@ubuntu-xenial $ make exercise_1
```

Now, execute the following:

```
vagrant@ubuntu-xenial $ ./FirstTest
```

Executing your first test (2)

Here is what you should see:

```
vagrant@ubuntu-xenial $ ./FirstTest
INFO: Running: Math_PowersOfTwo from FirstTest.cpp(7)
INFO: Passed: Math_PowersOfTwo
```

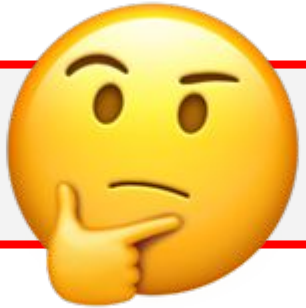
Our tests passed! This function must be correct, right?

Back to our test (1)

Here is what you will see inside of FirstTest.cpp

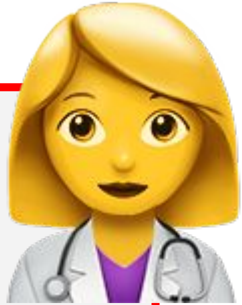
```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!

  ASSERT_EQ(Pow2(65535), 4294836225);
}
```

What does this do?



Back to our test (2)

Let's diagnose it!



We asked if this was true:

```
65535 * 65535 = 4294836225
```

We can express this in hexadecimal as:

```
0xFFFF * 0xFFFF = 0xFFFE_0001
```

And only the `0x0001` fits into a `uint16_t`

Back to our test (3)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!

  ASSERT_EQ(Pow2(65535), 1);
}
```

“correct”

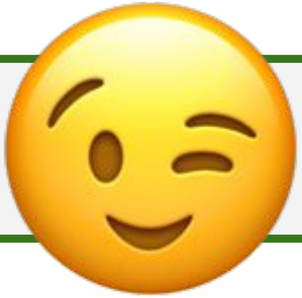
Back to our test

(4)

Here is what you will see inside of FirstTest.cpp

```
#include <deepstate/DeepState.hpp>

uint32_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!

  ASSERT_EQ(Pow2(65535), 4294836225);
}
```

Fixed!



Unit testing is great... right?

- **Unit tests help you to...**
  - Find bugs in your code
  - Experimentally verify your code on some set of inputs
  - Verify that the behavior of some code on some set of inputs stays consistent over time and across changes
- **But, unit tests are not a panacea**
  - It is up to YOU, the tester, to understand and test the boundary conditions, and test for them
  - This is harder for more complex code

Can’t we just automate it?

- **Ideally, we’d like something to figure out the best set of inputs for a given test so we don’t have to** (think so hard)
  - Spoiler alert! DeepState is that system
- **This is a “solved” problem**
  - Symbolic execution (e.g. KLEE, Manticore, Angr, S2E, etc.)
  - Fuzzers (e.g. libFuzzer, AFL, Dr. Fuzz, Radamsa, zzuf, Peach, etc.)
- **Developers don’t use existing solutions because they don’t fit nicely into their existing workflow!**

Developers don’t use security testing tools

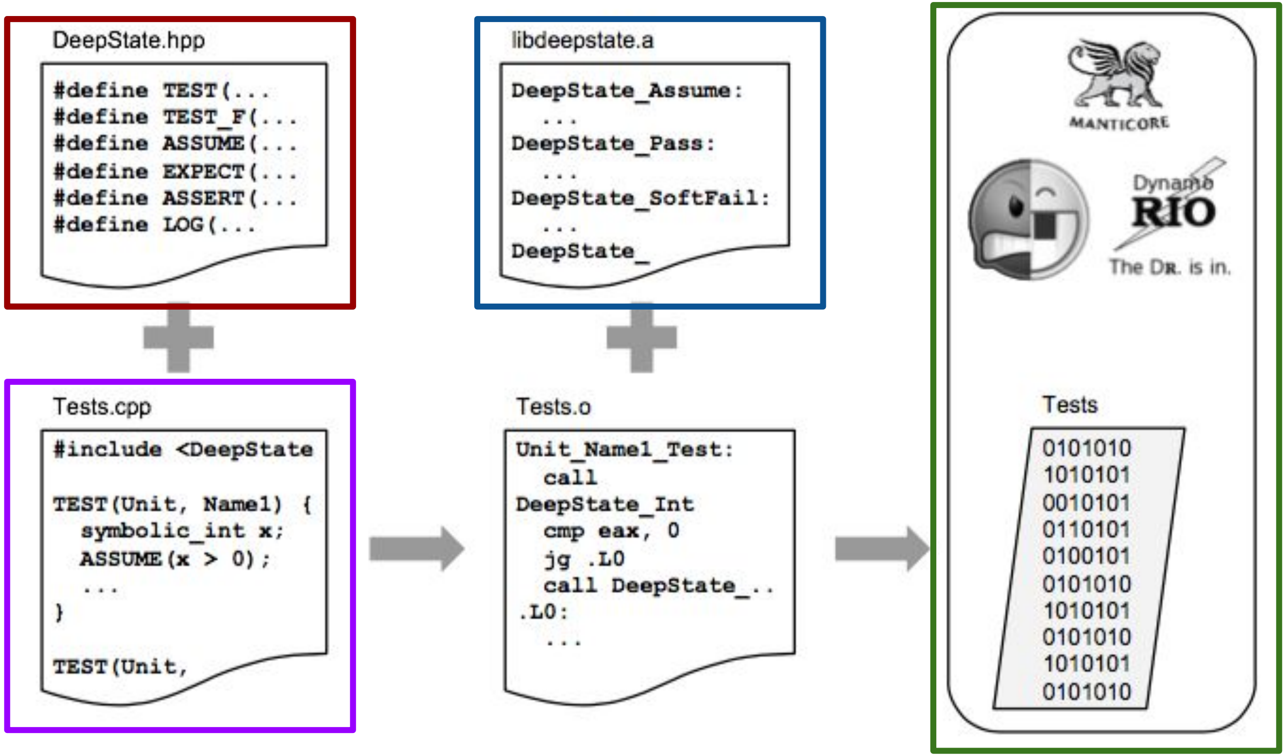
- **Zero* developers use symbolic executors**
  - Hard to learn and use
  - Difficult to integrate into a build/test cycle
  - Confusing and easily crash/run forever/eat up memory
- **Nearly zero* developers use fuzzers**
  - Requires custom harnesses and build system changes
- **Security tools are built for bug hunters**
  - Work great for auditors, CTF contests, reverse engineers
  - Confusing and alien for software developers

Developers do use unit testing

- **DeepState integrates symbolic testing and fuzz testing into a Google Test-like unit testing framework**
    - Fits into existing developer workflow
    - Easily integrates with existing code base and build system
    - Easy to learn and use, especially if you are familiar with Google Test
- **Improves software quality**
    - Also tests for correctness, not just security
    - No false positives!

Integrating DeepState is easy

- Header
- Library
- Test cases
- Executor



Writing unit tests with DeepState

- **TEST, TEST_F**
  - `TEST(UnitName, CaseName)` creates a new test
  - `TEST_F` is like `TEST` but with a class that performs setup and teardown
- **ASSERT, CHECK**
  - `ASSERT` logs and error and stops execution if a condition fails
  - `CHECK` is like `ASSERT` but logs an error and continues execution
- **Examples:**
  - `ASSERT(poly != y * z); ASSERT_NE(poly, y * z);`

Monitoring test progress in DeepState

- **Logging in unit tests is valuable for monitoring progress, debugging unusual outcomes**
- **Examples:**
  - `LOG(WARNING) << “hello” << “world!”;`
  - `ASSERT(true) << “Never printed because true is true”;`
  - `ASSERT(false) << “Always printed, test stops”;`
  - `CHECK(false) << “Always printed, test marked as ” << “failing but continues”;`

Creating “symbolic values” with DeepState

- **Symbolic data types**
  - Convenient typedefs: `symbolic_int`, `symbolic_char`, …
  - Explicit form: `Symbolic<int>`, `Symbolic<std::string>`, …
- **Constraining symbolic values**
  - `ASSUME`, `ASSUME_*` macros add constraints onto symbolic values, e.g. ensuring a value falls within a range
- **Examples:**
  - `symbolic_unsigned x, y, z;`
  - `ASSUME_GT(x, 0); ASSUME_GT(y, 1); ASSUME_GT(z, 1);`

Discovering the original bug with DeepState (1)

Here is what FirstTest.cpp looked like *before* our fix:

```
#include <deepstate/DeepState.hpp>

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0
  ASSERT_NE(Pow2(2), 3);  // 2^2 != 3

  // Try some for yourself!

}
```

Discovering the original bug with DeepState (2)

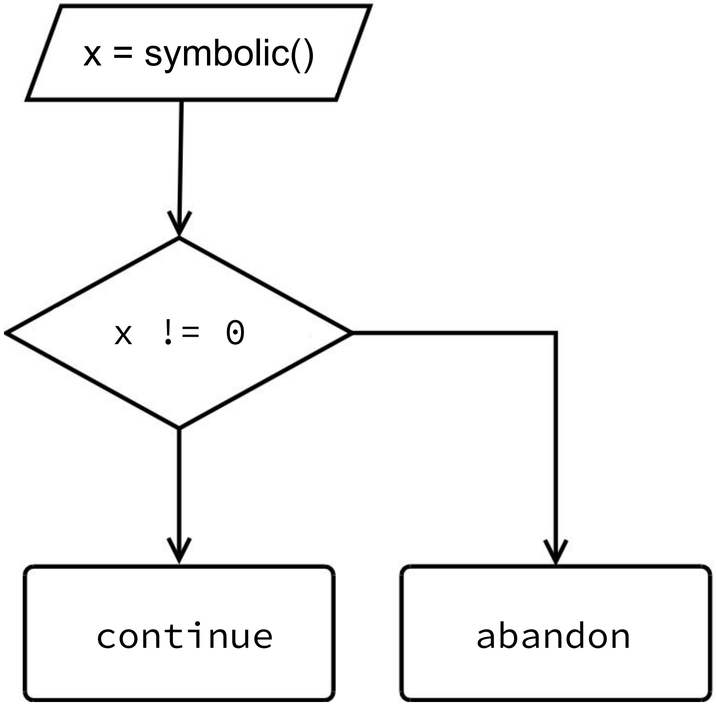
Here is how to use DeepState to discover the bug:

```
#include <deepstate/DeepState.hpp>

using namespace deepstate;

uint16_t Pow2(uint16_t x) {
  return x * x;
}

TEST(Math, PowersOfTwo) {
  ASSERT_EQ(Pow2(0), 0);  // 0^2 == 0

  Symbolic<uint16_t> x;
  ASSUME_NE(x, 0);
  ASSERT_EQ(Pow2(x) / x, x)  // forall x. (x^2)/x == x
      << "Pow2(" << x << ") / " << x << " != " << x;
}
```

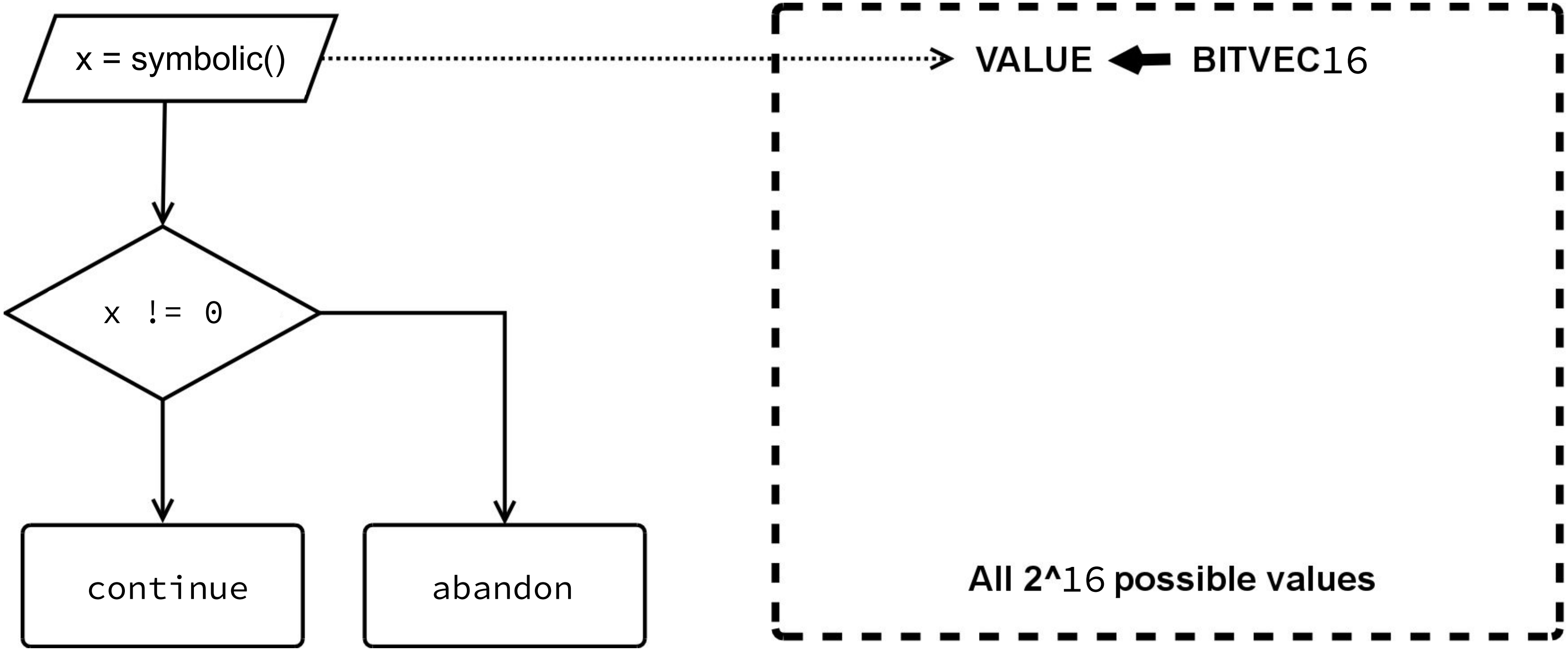
Discovering the original bug with DeepState (3)

```
vagrant@ubuntu-xenial $ deepstate-angr ./FirstTest

Running Math_PowersOfTwo from FirstTest.cpp(7)
…
FirstTest.cpp(11): Checked condition
FirstTest.cpp(12): Pow2(258) / 258 != 258
Failed: Math_PowersOfTwo
Input: 01 02
Saving input to out/FirstTest.cpp/Math_PowersOfTwo/0cb988d042a7f28dd5fe2b55b3f5ac7a.fail
Running Math_PowersOfTwo from FirstTest.cpp(7)
FirstTest.cpp(11): Checked condition
FirstTest.cpp(12): Pow2(256) / 256 != 256
Failed: Math_PowersOfTwo
Input: 01 00
Saving input to out/FirstTest.cpp/Math_PowersOfTwo/25daad3d9e60b45043a70c4ab7d3b1c6.fail
```

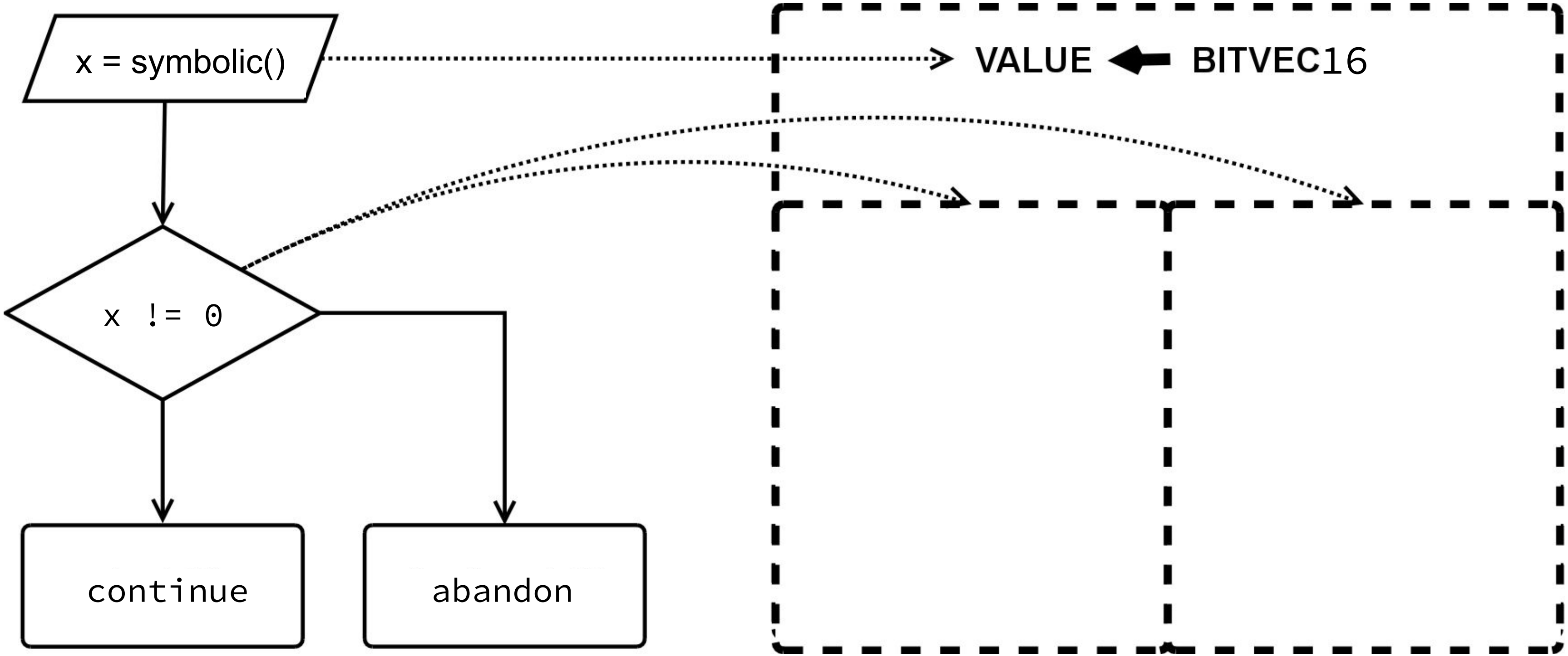
How did it do that? (1)



How did it do that? (2)

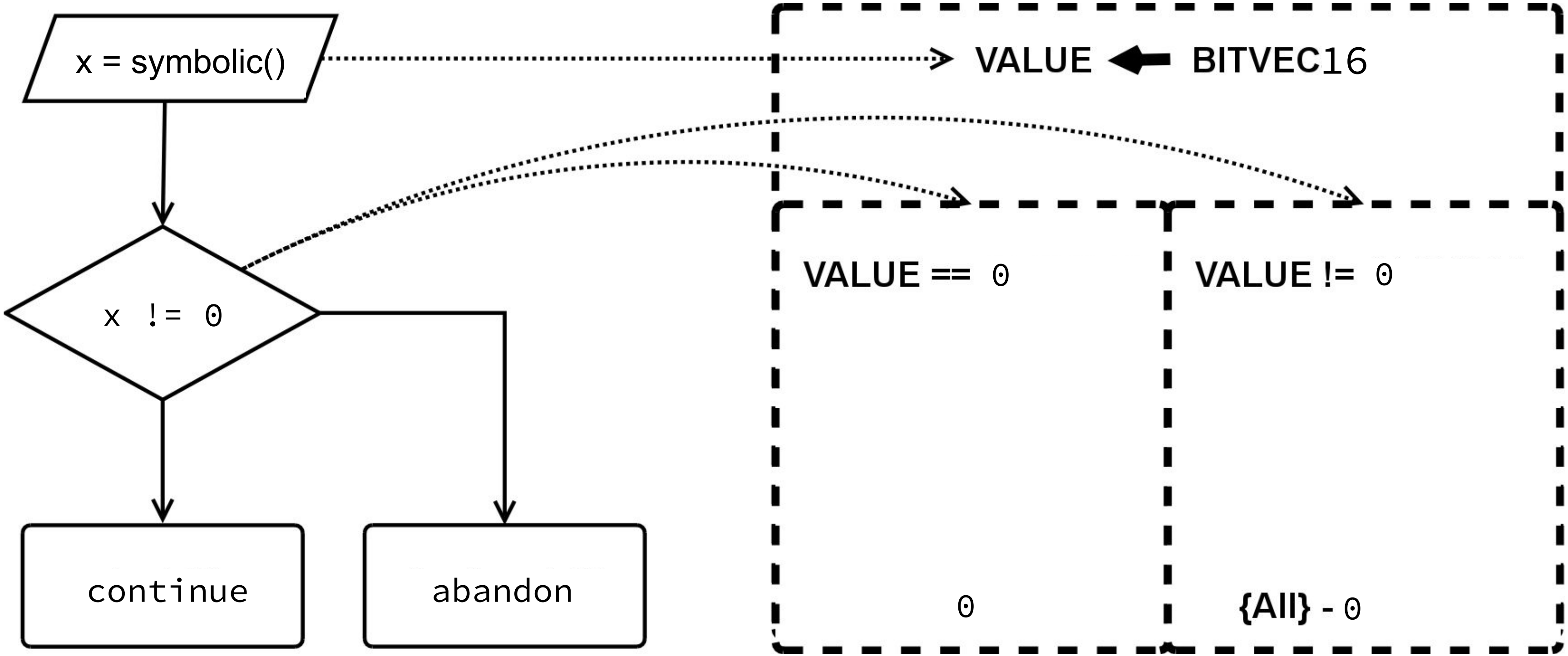


How did it do that? (3)



How did it do that?

(4)



Exercise 1.1 (1)

Enter the exercises directory and open LongLongOver.cpp

```
vagrant@ubuntu-xenial $ cd exercises
vagrant@ubuntu-xenial $ nano LongLongOver.cpp
```

To compile it, execute the following command:

```
vagrant@ubuntu-xenial $ make exercise_1.1
```

Exercise 1.1

(1)

Write a symbolic unit test for **`overflow_ll_add`** for <u>non negatives</u> x and y:

1. `overflow_ll_add(x,y)==0` ⇒ `x+y` does not overflow
2. `overflow_ll_add(x,y)==1` ⇒ `x+y` overflows

Write a DeepState test for (1) and test it. Then, write a DeepState test for (2) and test it.

Exercise 1.1

(2)

```
#include <deepstate/DeepState.hpp>
using namespace deepstate;

TEST(Math, NoOverflowAdd) {
  Symbolic<long long> x, y;
  // Fill me in!!!
  // Fill me in!!!
  // Fill me in!!!
  // Fill me in!!!
  // Fill me in!!!
}
```

Exercise 1.1

(3)

```
#include <deepstate/DeepState.hpp>
using namespace deepstate;

TEST(Math, NoOverflowAdd) {
  Symbolic<long long> x, y;
  // Your goals:
  //  1)  x and y should be non-negative
  //  2)  if overflow_ll_add of x and y doesn't overflow,
  //      then verify that the result of the addition, z,
  //      is greater than or equal to each of x and y
}
```

Exercise 1.1 (4)

```
#include <deepstate/DeepState.hpp>
using namespace deepstate;

TEST(Math, NoOverflowAdd) {
 Symbolic<long long> x, y;

 ASSUME_GE(x, 0);
 ASSUME_GE(y, 0);
 ASSUME_EQ(overflow_ll_add(x, y), 0);

 long long z = x + y;

 ASSERT(z >= x && z >= y);
}
```

Exercise 1.1

(4)

```
#include <deepstate/DeepState.hpp>
using namespace deepstate;

TEST(Math, OverflowAdd) {
 Symbolic<long long> x, y;

 ASSUME_GE(x, 0);
 ASSUME_GE(y, 0);
 ASSUME_EQ(overflow_ll_add(x, y), 1);

 long long z = x + y;

 ASSERT(z < x || z < y);
}
```

Exercise 1.1

(5)

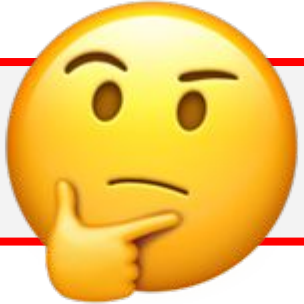
```
Running Math_NoOverflowAdd from LongLongOver.cpp(134)
...
Input: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 7f
Running Math_NoOverflowAdd from LongLongOver.cpp(134)
...
Passed: Math_OverflowAdd
Input: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Saving input to
out/LongLongOver.cpp/Math_NoOverflowAdd/4ae71336e44bf9bf79d2752e234818a5.pass
Running Math_NoOverflowAdd from LongLongOver.cpp(134)
...
Passed: Math_OverflowAdd
Input: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Saving input to
out/LongLongOver.cpp/Math_NoOverflowAdd/cf404dc806178c245b5b4fe2531e6d8c.pass
```

Exercise 1.1

(6)

```
Running Math_OverflowAdd from LongLongOver.cpp(150)
LongLongOver.cpp(154): Checked condition
LongLongOver.cpp(155): Checked condition
LongLongOver.cpp(156): Checked condition
LongLongOver.cpp(161): Checked condition
Failed: Math_OverflowAdd
Input: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 7f
Saving input to
out/LongLongOver.cpp/Math_OverflowAdd/1288b4cdc66d265fd60d3b52172ba717.fail
```

Why?

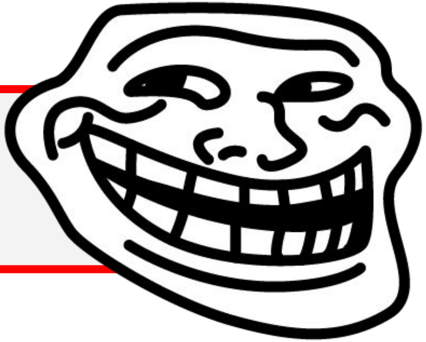


Exercise 1.1

(7)

```
#include <deepstate/DeepState.hpp>
using namespace deepstate;

TEST(Math, OverflowAdd) {
 Symbolic<long long> x, y;

 ASSUME_GE(x, 0);
 ASSUME_GE(y, 0);
 ASSUME_EQ(overflow_ll_add(x, y), 1);

 long long z = x + y;

 ASSERT(z < x || z < y);
}
```

undefined behavior



Exercise 1.1

(8)

```
#include <deepstate/DeepState.hpp>
using namespace deepstate;

TEST(Math, OverflowAdd) {
 Symbolic<long long> x, y;

 ASSUME_GE(x, 0);
 ASSUME_GE(y, 0);
 ASSUME_EQ(overflow_ll_add(x, y), 1);

 volatile long long z = x + y;

 ASSERT(z < x || z < y);
}
```

Exercise 1.1

(9)

```
Running Math_OverflowAdd from LongLongOver.cpp(150)
LongLongOver.cpp(154): Checked condition
LongLongOver.cpp(155): Checked condition
LongLongOver.cpp(156): Checked condition
Passed: Math_OverflowAdd
Input: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 7f
Saving input to
out/LongLongOver.cpp/Math_OverflowAdd/1288b4cdc66d265fd60d3b52172ba717.pass
```

Exercise 2 (1)

For the next example, execute the following command:

```
vagrant@ubuntu-xenial $ make exercise_2
```

Now, execute the following:

```
vagrant@ubuntu-xenial $ ./Wallet
```

Exercise 2 (2)

Here is what you should see:

```
vagrant@ubuntu-xenial $ ./Wallet

Usage: ./Wallet <initial_balance> W|D <amount> [W|D <amount> [...]]
```

Wallet.hpp implementation (1)

```
class Wallet;

struct Cheque {
  unsigned amount;
  Wallet *dest;
};

class Wallet {
 public:
  Wallet(void)
      : balance(0) {}

  explicit Wallet(unsigned initial_balance)
      : balance(initial_balance) {}

  void Deposit(unsigned amount) {
    balance += amount;
  }
  …

 private:
  unsigned balance;
```

Wallet.hpp implementation (2)

```
unsigned Balance(void) const {
  return balance;
}

bool Withdraw(unsigned amount) {
  if (amount <= balance) {
    balance -= amount;
    return true;
  } else {
    return false;
  }
}

bool Transfer(Cheque cheque) {
  if (Withdraw(cheque.amount)) {
    cheque.dest->Deposit(cheque.amount);
    return true;
  } else {
    return false;
  }
}
…
```

Wallet.hpp implementation (3)

```
bool MultiTransfer(const std::vector<Cheque> &cheques) {

  LOG(DEBUG)
      << "Processing " << cheques.size() << " cheques";

  unsigned total_to_withdraw = 0;
  for (auto cheque : cheques) {
    total_to_withdraw += cheque.amount;
  }

  if (balance < total_to_withdraw) {
    LOG(WARNING)
        << "Insufficient funds! Can't transfer " << total_to_withdraw
        << " from account with balance of " << balance;
    return false;
  }

  LOG(DEBUG)
      << "Withdrawing " << total_to_withdraw << " from account";

  for (auto cheque : cheques) {
    ASSERT(Transfer(cheque))
        << "Insufficient funds! Can't transfer " << cheque.amount
        << " from account with balance of " << balance;
  }

  return true;
}
```

Exercise 2: Testing Wallet.hpp

Write DeepState test cases to test the functionality of Wallet:

1. A valid withdrawal decreases the account balance
2. A failed withdrawal preserves the account balance
3. A self-transfer preserves the account balance
4. A multi transfer preserves the total balance between two accounts.

Write DeepState tests for 1, 2, and 3 and execute them with `deepstate-angr`. Then, write a DeepState test for 4 and execute it as well.

Wallet_tests.cpp test fixture

```cpp
class WalletTests : public deepstate::Test {
 public:
  WalletTests(void)
      : account1(initial_balance1),
        account2(initial_balance2) {}

  uint32_t InitialBalance(void) const {
    return initial_balance1 + initial_balance2;
  }

  uint32_t TotalBalance(void) const {
    return account1.Balance() + account2.Balance();
  }
 protected:

  symbolic_unsigned initial_balance1;
  symbolic_unsigned initial_balance2;

  Wallet account1;
  Wallet account2;

  symbolic_unsigned amount1;
  symbolic_unsigned amount2;
};
```

Wallet tests using the WalletTest fixture

```
TEST_F(WalletTests, WithdrawalDecreasesAccountBalance) {
  // Fill me in!!!
}

TEST_F(WalletTests, FailedWithdrawalPreservesAccountBalance) {
  // Fill me in!!!
}

TEST_F(WalletTests, SelfTransferPreservesAccountBalance) {
  // Fill me in!!!
}

TEST_F(WalletTests, MultiTransferPreservesBankBalance) {
  // Fill me in!!!
}
```

Withdrawal and transfer properties (1)

```
TEST_F(WalletTests, WithdrawalDecreasesAccountBalance) {
  ASSUME_GT(amount1, 0);
  ASSUME(account1.Withdraw(amount1));
  ASSERT_LT(account1.Balance(), initial_balance1);
}

TEST_F(WalletTests, FailedWithdrawalPreservesAccountBalance) {
  …
}

TEST_F(WalletTests, SelfTransferPreservesAccountBalance) {
  …
}
```

Withdrawal and transfer properties (2)

```
TEST_F(WalletTests, WithdrawalDecreasesAccountBalance) {
  ASSUME_GT(amount1, 0);
  ASSUME(account1.Withdraw(amount1));
  ASSERT_LT(account1.Balance(), initial_balance1);
}

TEST_F(WalletTests, FailedWithdrawalPreservesAccountBalance) {
  ASSUME(!account1.Withdraw(amount1));
  ASSERT_EQ(account1.Balance(), initial_balance1);
}

TEST_F(WalletTests, SelfTransferPreservesAccountBalance) {
  …
}
```

Withdrawal and transfer properties (3)

```
TEST_F(WalletTests, WithdrawalDecreasesAccountBalance) {
  ASSUME_GT(amount1, 0);
  ASSUME(account1.Withdraw(amount1));
  ASSERT_LT(account1.Balance(), initial_balance1);
}

TEST_F(WalletTests, FailedWithdrawalPreservesAccountBalance) {
  ASSUME(!account1.Withdraw(amount1));
  ASSERT_EQ(account1.Balance(), initial_balance1);
}

TEST_F(WalletTests, SelfTransferPreservesAccountBalance) {
  (void) account1.Transfer({amount1, &account1});

  ASSERT_EQ(account1.Balance(), initial_balance1)
      << "Account1's balance has changed with a self transfer of "
      << amount1;
}
```

Multi-transfer property

```
TEST_F(WalletTests, MultiTransferPreservesBankBalance) {
  const auto old_balance1 = account1.Balance();
  const auto old_balance2 = account2.Balance();

  const auto transfer_succeeded = account1.MultiTransfer({
    {amount1, &account2},
    {amount2, &account2},
  });

  if (!transfer_succeeded) {
    CHECK(old_balance1 == account1.Balance())
        << "Account1's balance has changed from "
        << old_balance1 << " to " << account1.Balance();

    CHECK(old_balance2 == account2.Balance())
        << "Account2's balance has changed from "
        << old_balance2 << " to " << account2.Balance();

  } else {
    CHECK(InitialBalance() == TotalBalance())
        << "Balance in bank has changed from "
        << InitialBalance() << " to " << TotalBalance();
  }
}
```

End of part 1

Welcome back

Summary of part 1

- **Unit testing is great, but making good unit tests is hard**
  - Easy to write tests
  - Just as easy to miss corner cases
- **DeepState turns unit testing into proving**
  - Instead of writing tests with specific inputs to test, use symbolic variables/values to test *for all* inputs

Overview of part 2

- **The leaky abstraction: symbolic executions tactics**
  - Helping to mitigate the “path explosion” problem
- **When one approach fails, try, try, try again**
  - We saw `deepstate-angr`, but there’s more than just that
  - Other input-finding backends: Manticore, AFL, libFuzzer, Dr. Fuzz, S2E
- **Time to get real**
  - Testing file system durability: is filesystem metadata consistent in the face of arbitrary shutdowns?

Sometimes abstractions leak through (1)

- **Symbolic execution is a powerful program analysis technique**
  - Explores *all* feasible paths through a program, but what does this mean, really?
  - If execution reaches an `if` statement, then a symbolic executor will try to discover (e.g. via a SMT theorem prover) inputs that drive execution down both paths
  - Any time a symbolic executor is faced with more than one possible paths to explore, it chooses to explore all of them (e.g. via enqueuing them)

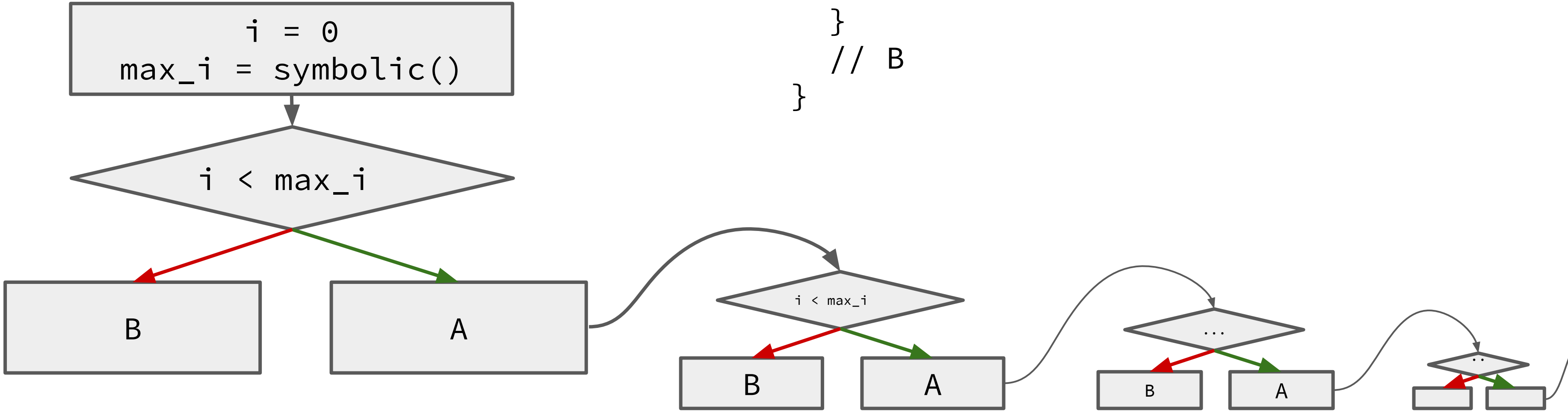
Sometimes abstractions leak through (2)

What if we have an for loop with a symbolic upper bound?

```
TEST(PathExplosion, GoesBoom) {
  symbolic_int max_i;
  for (int i = 0; i < max_i; ++i) {
    // A
  }
  // B
}
```

Sometimes abstractions leak through (3)

```
TEST(PathExplosion, GoesBoom) {
  symbolic_int max_i;
  for (int i = 0; i < max_i; ++i) {
    // A
  }
  // B
}
```



Sometimes abstractions leak through (4)

- **Symbolic upper bounds to loops can cause unbounded forking**
  - Every iteration will cause the symbolic executor to explore both paths
  - Imagine if there was a nested loop, or an if statement in the loop
- **These constructs are common in real code**
  - Need a way to mitigate the path explosion
  - Solution: sacrifice some generality to get performance by “pre-forking” and unrolling the loops in each fork

Sometimes abstractions leak through (5)

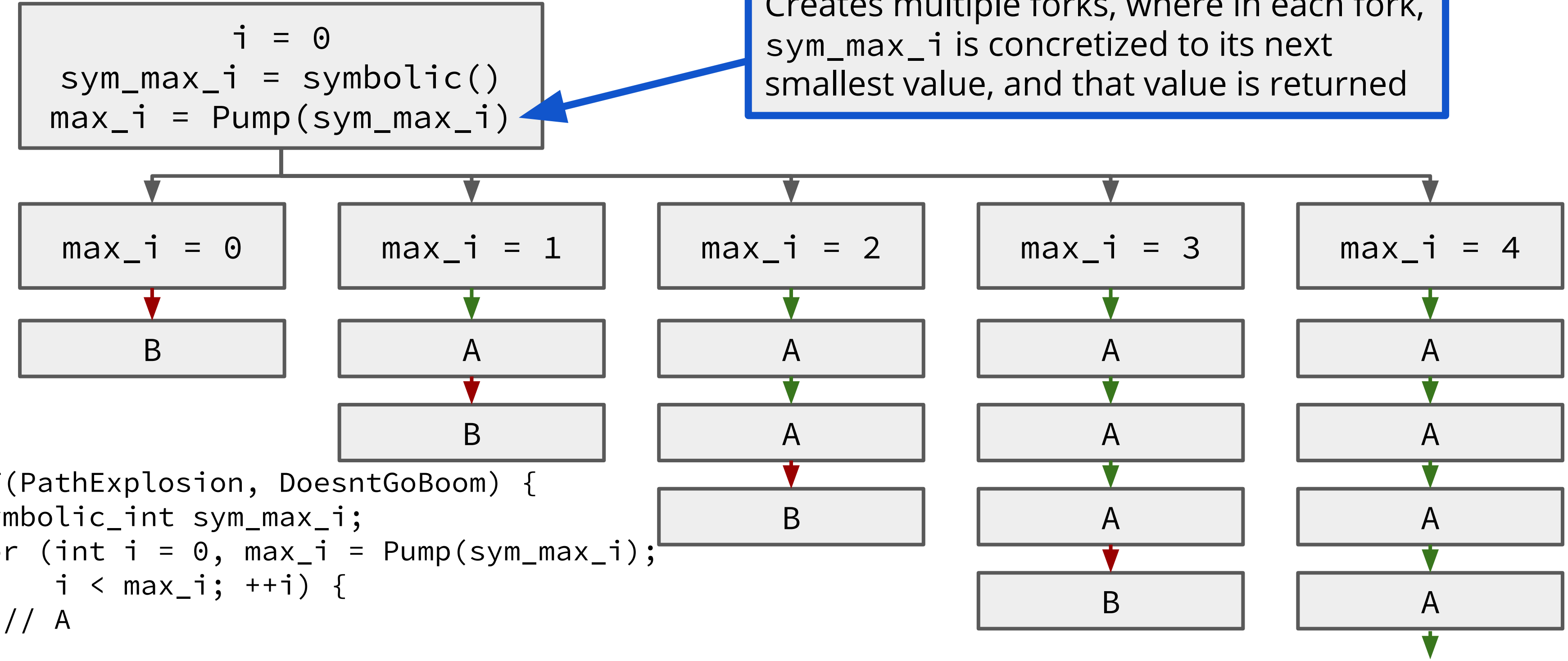
With the "pumping" tactic of gener

```
TEST(PathExplosion, DoesntGoBoom) {
  symbolic_int sym_max_i;
  for (int i = 0, max_i = Pump(sym_max_i);
       i < max_i; ++i) {
    // A
  }
  // B
}
```

Not symbolic!

Sometimes abstractions leak through (6)

Creates multiple forks, where in each fork, sym_max_i is concretized to its next smallest value, and that value is returned




```
TEST(PathExplosion, DoesntGoBoom) {
  symbolic_int sym_max_i;
  for (int i = 0, max_i = Pump(sym_max_i);
       i < max_i; ++i) {
    // A
  }
  // B
}
```

Sometimes abstractions leak through (7)

- **Pumping is one way to mitigate path explosion in symbolic execution**
  - Perhaps a better name would be “`MinPump`” or “`MinValues`”
  - Arbitrary policies are possible, e.g. `MaxPump`, `MinMaxPump`, etc.
- **Idiom exists to improve scalability of symbolic execution**
  - Usage of this idiom tends toward concretizing loop upper bounds
  - This is a useful semantic to “attach onto” for test case reduction
- **But what if none of these idioms “solve” path explosion?**

But what if we can’t mitigate path explosion? (1)

- **Sometimes we can’t easily mitigate path explosion with idioms/tactics like Pump**
  - No fear, libFuzzer is here!
- **DeepState supports multiple input-generation backends**
  - Manticore, Angr, AFL, libFuzzer, AFL, Dr. Fuzz, and S2E
  - If one doesn’t work or is too slow, try another!

But what if we can't mitigate path explosion? (2)

- **Fuzzers (e.g. libFuzzer, AFL) can be really effective at finding the inputs that trigger the unusual cases**
  - Instead of using a symbolic executor and having it reason over paths, we use a code coverage or "data coverage" guided fuzzer to brute force the inputs
  - Tends to be faster than symbolic executors, works for some cases where the symbolic executors do not (e.g. testfs)

Let's get real: file systems

Alex Groce talks about file system testing at NASA, JPL, and how we're using DeepState to test https://github.com/agroce/testfs

TRAIL OF BITS


TRAIL OF BITS