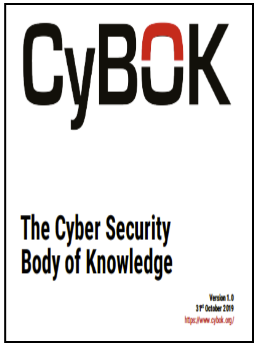
Systems and Software Verification Laboratory

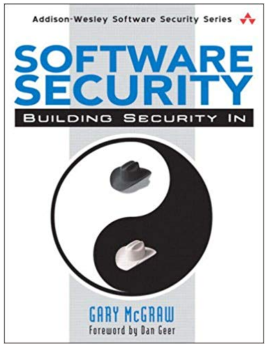
Detection of Software Vulnerabilities: Dynamic Analysis

Lucas Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk

Dynamic Analysis

- Lucas Cordeiro (Formal Methods Group)
  - lucas.cordeiro@manchester.ac.uk
  - Office: 2.28
  - Office hours: 15-16 Tuesday, 14-15 Wednesday
- References:
  - *Software Security: Building Security In* (Chapter 6)
  - *Automated Whitebox Fuzz Testing* by Godefroid et al.
  - *The Cyber Security Body of Knowledge* by Rashid et al.
  - *Security Testing* by Erik Poll





Intended learning outcomes

- Understand **dynamic detection techniques** to identify security vulnerabilities

Intended learning outcomes

- Understand **dynamic detection techniques** to identify security vulnerabilities
- Generate **executions of the program** along paths that will lead to the **discovery of new vulnerabilities**

Intended learning outcomes

- Understand **dynamic detection techniques** to identify security vulnerabilities
- Generate **executions of the program** along paths that will lead to the **discovery of new vulnerabilities**
- Explain **black-box fuzzing: grammar-based** and **mutation-based fuzzing**

Intended learning outcomes

- Understand **dynamic detection techniques** to identify security vulnerabilities
- Generate **executions of the program** along paths that will lead to the **discovery of new vulnerabilities**
- Explain **black-box fuzzing: grammar-based** and **mutation-based fuzzing**
- Explain **white-box fuzzing: dynamic symbolic execution**

Intended learning outcomes

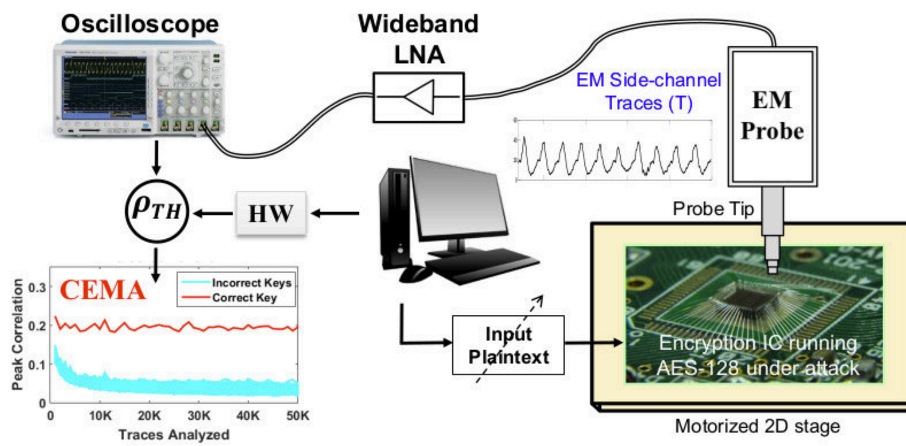
- Understand **dynamic detection techniques** to identify security vulnerabilities
- Generate **executions of the program** along paths that will lead to the **discovery of new vulnerabilities**
- Explain **black-box fuzzing: grammar-based** and **mutation-based fuzzing**
- Explain **white-box fuzzing: dynamic symbolic execution**

Security in the Development Lifecycle

- A majority of **security defects** and vulnerabilities in software are **not directly related to functionality**

Security in the Development Lifecycle

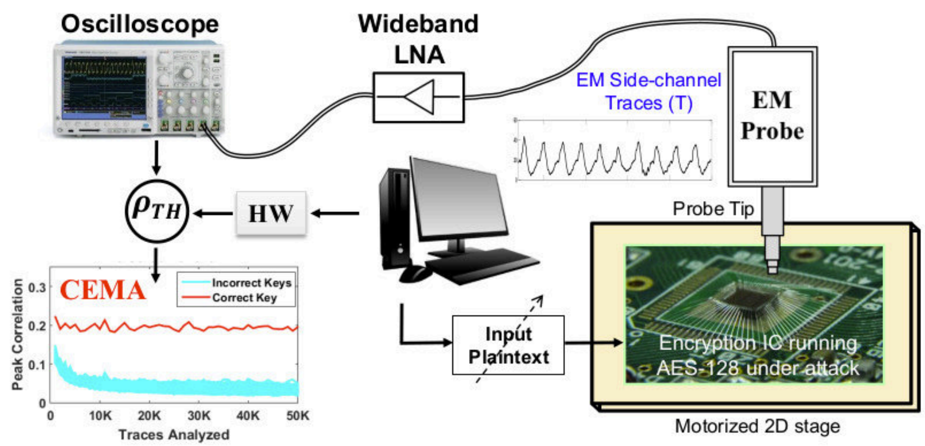
- A majority of **security defects** and vulnerabilities in software are **not directly related to functionality**
- **Side-channel effect in the hardware**
  - information obtained from the impl. rather than weaknesses in the code



STELLAR: A Generic EM Side-Channel Attack Protection through Ground-Up Root-cause Analysis, HOST2019.

Security in the Development Lifecycle

- A majority of **security defects** and vulnerabilities in software are **not directly related to functionality**
- **Side-channel effect in the hardware**
  - information obtained from the impl. rather than weaknesses in the code



STELLAR: A Generic EM Side-Channel Attack Protection through Ground-Up Root-cause Analysis, HOST2019.

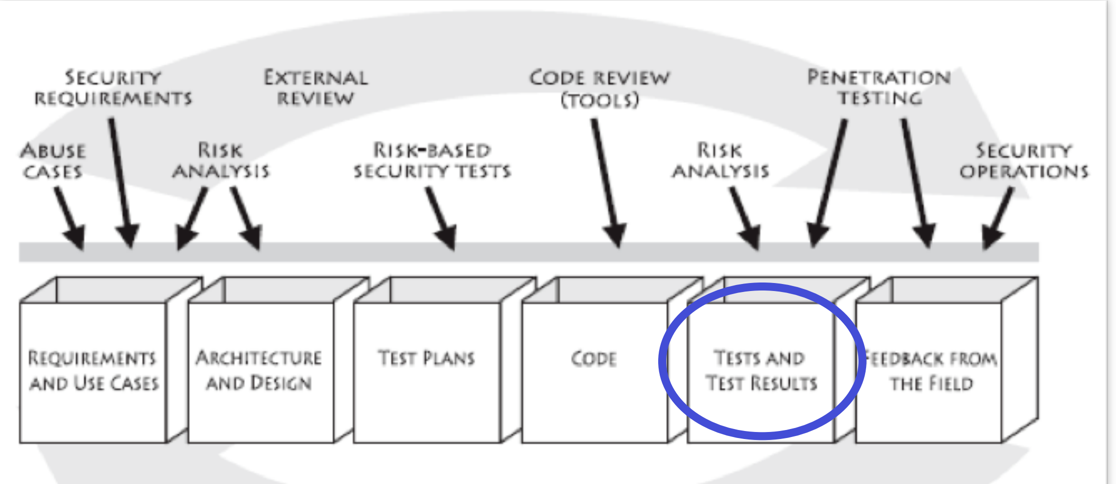
timing information and power consumption can be exploited

Security in the Development Lifecycle

- **Security testing:** white hat, red hat, and penetration

Security in the Development Lifecycle

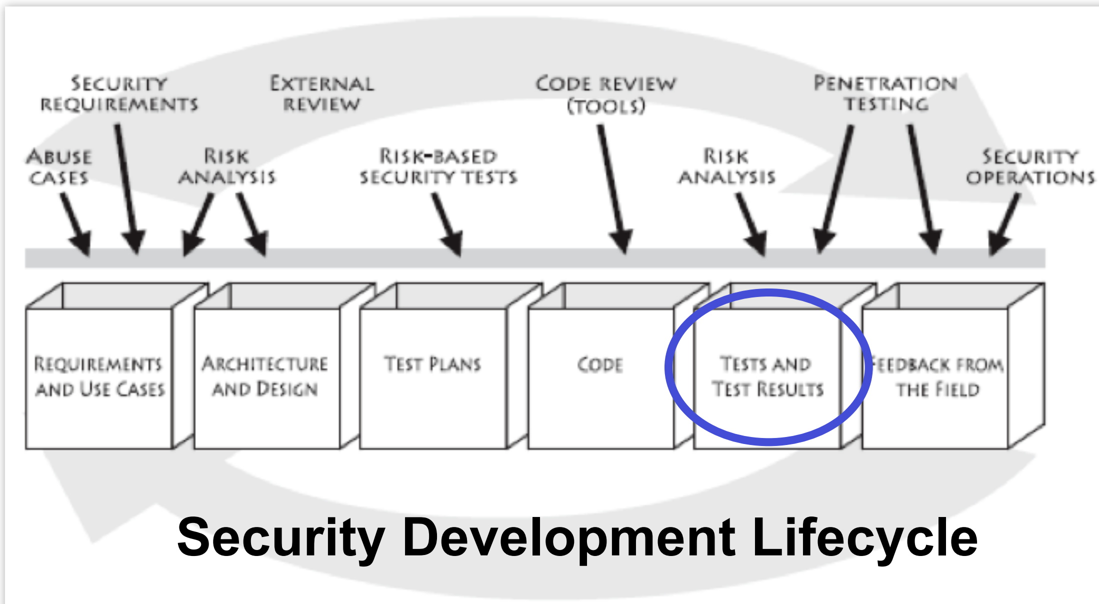
- **Security testing:** white hat, red hat, and penetration



Security Development Lifecycle

Security in the Development Lifecycle

- **Security testing:** white hat, red hat, and penetration



- **Testing for a negative** poses a much greater challenge than **verifying for a positive**

Testing for functionality vs testing for security

- **Traditional testing** checks **functionalities** for **sensible inputs** and **corner conditions**

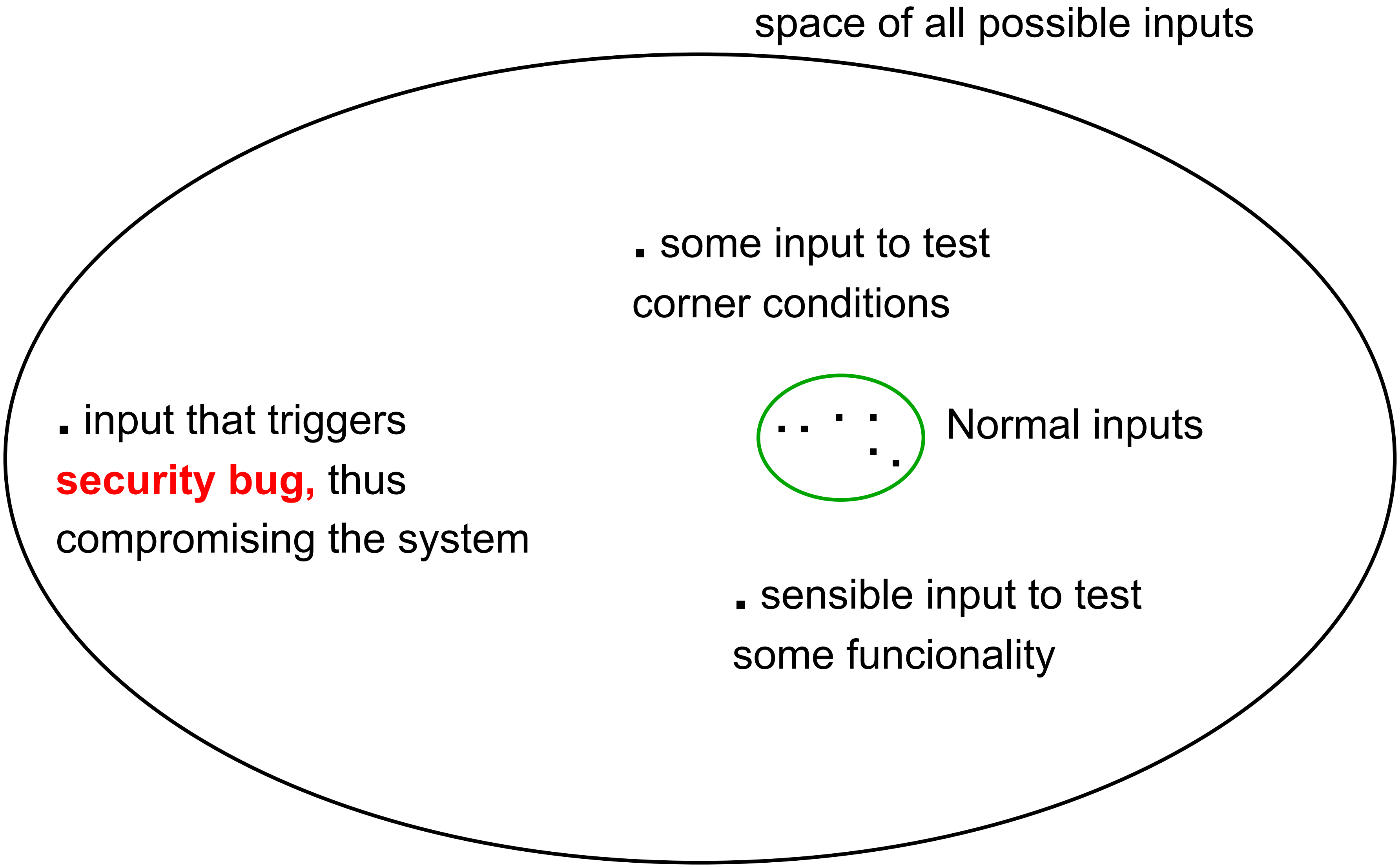
Testing for functionality vs testing for security

- **Traditional testing** checks **functionalities** for **sensible inputs** and **corner conditions**
- **Security testing** also requires looking for the wrong, unwanted behavior for **uncommon inputs**

Testing for functionality vs testing for security

- **Traditional testing** checks **functionalities** for **sensible inputs** and **corner conditions**
- **Security testing** also requires looking for the wrong, unwanted behavior for **uncommon inputs**
- **Routine use** of a software system is more likely to reveal **functional problems** than **security problems**:
  - **users** will complain about **functional problems**, but **hackers** will not complain about **security problems**

Security testing is difficult



Definition of Test Suite and Oracle

- To test a software system, we need:

① **test suite:** a collection of input data

② **test oracle:** decides if a test succeeded or led to an error

   - some way to decide if the software behaves as we want

Definition of Test Suite and Oracle

- To test a software system, we need:

① **test suite:** a collection of input data

② **test oracle:** decides if a test succeeded or led to an error

  - some way to decide if the software behaves as we want

- Define both test suites and test oracles can be a **significant work**
  - A test oracle consists of a long list, which **for every individual test case, specifies what should happen**
  - A **simple test oracle**: just looking if the application does not crash

Statement Coverage

- **Statement coverage** involves the execution of all the executable statements at least once
  - (executed statements / total statements)*100

```c
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    *res = 0;
    return 1;
  } else if(a == 1) {
    *res = b;
    return 1;
  } else if(b == 1) {
    *res = a;
    return 1;
  }
  *res = a * b; // there exists an overflow
  return 1;
}
```

Statement Coverage

- **Statement coverage** involves the execution of all the executable statements at least once
  - (executed statements / total statements)*100

```
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    *res = 0;
    return 1;
  } else if(a == 1) {
    *res = b;
    return 1;
  } else if(b == 1) {
    *res = a;
    return 1;
  }
  *res = a * b; // there exists an overflow
  return 1;
}
```

a=0,b=0

Coverage=3/11=27%

Statement Coverage

- **Statement coverage** involves the execution of all the executable statements at least once
  - (executed statements / total statements)*100

```c
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    *res = 0;
    return 1;
  } else if(a == 1) {
    *res = b;
    return 1;
  } else if(b == 1) {
    *res = a;
    return 1;
  }
  *res = a * b; // there exists an overflow
  return 1;
}
```

a=1,b=3
Coverage=4/11=36%

Statement Coverage

- **Statement coverage** involves the execution of all the executable statements at least once
  - (executed statements / total statements)*100

```c
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    *res = 0;
    return 1;
  } else if(a == 1) {
    *res = b;
    return 1;
  } else if(b == 1) {
    *res = a;
    return 1;
  }
  *res = a * b; // there exists an overflow
  return 1;
}
```

a=2,b=1
Coverage=5/11=45%

Statement Coverage

- **Statement coverage** involves the execution of all the executable statements at least once
  – (executed statements / total statements)*100

```
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    *res = 0;
    return 1;
  } else if(a == 1) {
    *res = b;
    return 1;
  } else if(b == 1) {
    *res = a;
    return 1;
  }
  *res = a * b; // there exists an overflow
  return 1;
}
```

a=2,b=2
Coverage=5/11=45%

Statement Coverage

- **Statement coverage** involves the execution of all the executable statements at least once
  - (executed statements / total statements)*100

Test Case	Value of "a"	Value of "b"	Value of "res"	Statement Coverage
1	0	0	0	27%
2	1	3	b	36%
3	2	1	a	45%
4	2	2	a * b	45%

Decision Coverage

- **Decision coverage** reports the true or false outcomes of each Boolean expression (tough to achieve 100%)
  - (decision outcomes exercised / total decision outcomes) * 100

Decision Coverage

- **Decision coverage** reports the true or false outcomes of each Boolean expression (tough to achieve 100%)
  - (decision outcomes exercised / total decision outcomes) * 100

```c
void Demo(int a) {
  if (a > 5)
     a = a*3;
  printf("a: %i"\n);
}
```

a=4

(a>5) is false

Decision coverage = 50%

Decision Coverage

- **Decision coverage** reports the true or false outcomes of each Boolean expression (tough to achieve 100%)
  – (decision outcomes exercised / total decision outcomes) * 100

```
void Demo(int a) {
  if (a > 5)
    a = a*3;
  printf("a: %i"\n);
}
```

a=10
(a>5) is **true**
Decision coverage = 50%

Decision Coverage

- **Decision coverage** reports the true or false outcomes of each Boolean expression (tough to achieve 100%)
  - (decision outcomes exercised / total decision outcomes) * 100

```
void Demo(int a) {
  if (a > 5)
    a = a*3;
  printf("a: %i"\n);
}
```

Test Case	Value of “a”	Output	Decision Coverage
1	4	4	50%
2	10	30	50%

Branch Coverage

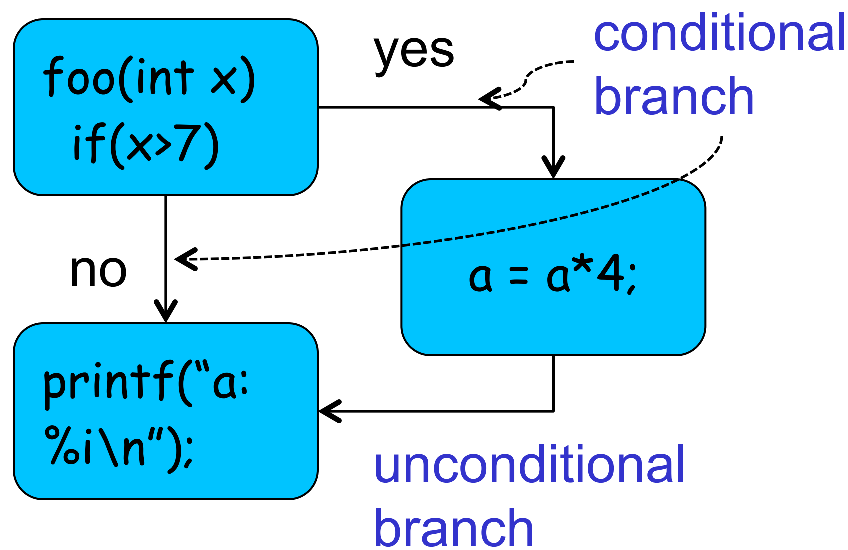
- **Branch coverage** tests every outcome from the code to ensure that every branch is executed at least once
  - (executed branches / total branches)*100

```
void foo(int x) {
  if (x > 7)
    a = a*4;
  printf("a: %i"\n);
}
```

Branch Coverage

- **Branch coverage** tests every outcome from the code to ensure that every branch is executed at least once
  - (executed branches / total branches)*100

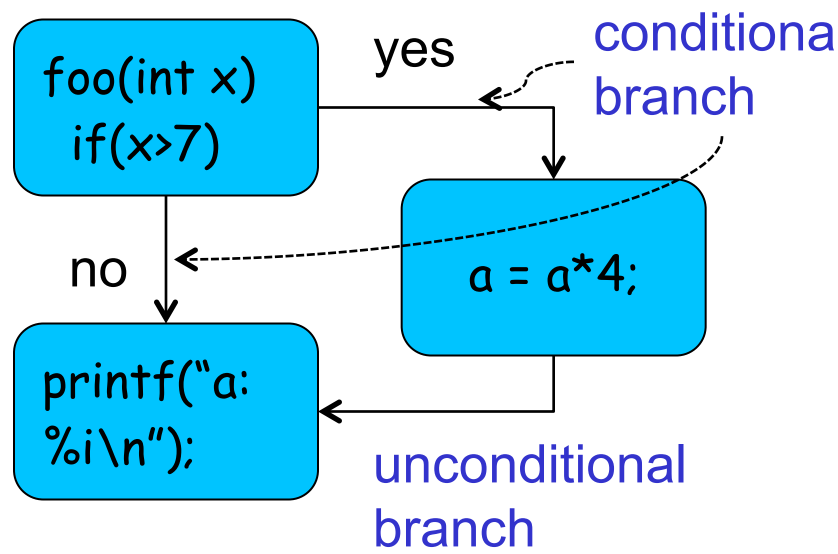
```
void foo(int x) {
  if (x > 7)
    a = a*4;
  printf("a: %i"\n);
}
```



Branch Coverage

- **Branch coverage** tests every outcome from the code to ensure that every branch is executed at least once
  - (executed branches / total branches)*100

```
void foo(int x) {
  if (x > 7)
    a = a*4;
  printf("a: %i"\n);
}
```



Test Case	Value of "a"	Output	Decision Coverage	Branch Coverage
1	4	4	50%	33%
2	10	40	50%	67%

Condition Coverage

- **Condition coverage** reveals how the variables in the conditional statement are evaluated (logical operands)
  - (executed operands / total operands)*100

```
int main() {
  unsigned int x, y, a, b;
  if((x < y) && (a>b))
    return 0;
  else
    return -1;
}
```

Condition Coverage

- **Condition coverage** reveals how the variables in the conditional statement are evaluated (logical operands)
  - (executed operands / total operands)*100

```c
int main() {
  unsigned int x, y, a, b;
  if((x < y) && (a>b))
    return 0;
  else
    return -1;
}
```

x<y	a>b	(x < y) && (a>b)
0	0	0
0	1	0
1	0	0
1	1	1

Condition Coverage

- **Condition coverage** reveals how the variables in the conditional statement are evaluated (logical operands)
  – (executed operands / total operands)*100

```
int main() {
  unsigned int x, y, a, b;
  if((x < y) && (a>b))
    return 0;
  else
    return -1;
}
```

x<y	a>b	(x < y) && (a>b)
0	0	0
0	1	0
1	0	0
1	1	1

Input	Condition	Outcome	Coverage
x=3, x=4	x<y	TRUE	25%
a=3, b=4	a>b	FALSE	25%

Code coverage criteria

- Code coverage criteria to measure the **test suite quality**
  - **Statement, decision, branch and condition coverage**

Code coverage criteria

- Code coverage criteria to measure the **test suite quality**
  - **Statement, decision, branch and condition coverage**
- Statement coverage does not imply branch coverage; e.g. for

```
void f (int a, int b) {
    if (a<100) {b--};
    a+=2;
}
```

Statement coverage needs 1 test case; branch coverage needs 2

Code coverage criteria

- Code coverage criteria to measure the **test suite quality**
  – **Statement, decision, branch and condition coverage**
- Statement coverage does not imply branch coverage; e.g. for

```
void f (int a, int b) {
  if (a<100) {b--};
  a+=2;
}
```

Statement coverage needs 1 test case; branch coverage needs 2

- Other coverage criteria exists, e.g., **modified condition/ decision coverage** (MCDC), which is used to test **avionics embedded software**

Modified condition/decision coverage (MC/DC)

- MC/DC coverage is similar to condition coverage, but we must **test every condition in a decision independently** to reach full coverage
- MC/DC requires all of the below during testing:
  - We invoke each entry and exit point
  - We test every possible outcome for each decision
  - Each condition in a decision takes every possible outcome
  - We show each condition in a decision to affect the outcome of the decision independently

Example of MC/DC

- Consider the following fragment of C code:

```
void foo(_Bool A, _Bool B, _Bool C) {
  if ( (A || B) && C ) {
    /* instructions */
  } else {
    /* instructions */
}
```

https://www.verifysoft.com/en_example_mcdc.html

Example of MC/DC

- Consider the following fragment of C code:

```c
void foo(_Bool A, _Bool B, _Bool C) {
  if ( (A || B) && C ) {
    /* instructions */
  } else {
    /* instructions */
}
```

- **Condition coverage:** A, B, and C should be evaluated at least one time “true” and one time “false”:
  - A = true / B = true / C = true
  - A = false / B = false / C = false

https://www.verifysoft.com/en_example_mcdc.html

Example of MC/DC

- Consider the following fragment of C code:

```c
void foo(_Bool A, _Bool B, _Bool C) {
  if ( (A || B) && C ) {
    /* instructions */
  } else {
    /* instructions */
}
```

- **Decision coverage:** the condition ((A || B) && C) should also be evaluated at least one time to “true” and one time to “false”:
  - A = true / B = true / C = true
  - A = false / B = false / C = false

https://www.verifysoft.com/en_example_mcdc.html

Example of MC/DC

- Consider the following fragment of C code:

```c
void foo(_Bool A, _Bool B, _Bool C) {
  if ( (A || B) && C ) {
    /* instructions */
  } else {
    /* instructions */
}
```

- MC/DC: each Boolean variable should be evaluated one time to “true” and one time to “false”, and this with affecting the decision's outcome

https://www.verifysoft.com/en_example_mcdc.html

Example of MC/DC

- Consider the following fragment of C code:

```c
void foo(_Bool A, _Bool B, _Bool C) {
  if ( (A || B) && C ) {
    /* instructions */
  } else {
    /* instructions */
}
```

- MC/DC: For a decision with *n* atomic boolean conditions, we have to find at least *n+1* tests

A = false / B = false / C = true ➔ evaluates to "false"
A = false / B = true / C = true ➔ evaluates to "true"
A = false / B = true / C = false ➔ evaluates to "false"
A = true / B = false / C = true ➔ evaluates to "true"

https://www.verifysoft.com/en_example_mcdc.html

Dynamic Detection

Dynamic detection techniques **execute a program** and **monitor the execution** to detect **vulnerabilities**

Dynamic Detection

Dynamic detection techniques **execute a program** and **monitor the execution** to detect **vulnerabilities**

- There exist two essential and relatively independent aspects of **dynamic detection**:
  - How should one **monitor an execution** such that vulnerabilities are detected?

Dynamic Detection

Dynamic detection techniques **execute a program** and **monitor the execution** to detect **vulnerabilities**

- There exist two essential and relatively independent aspects of **dynamic detection**:
  - How should one **monitor an execution** such that vulnerabilities are detected?
  - **How many and what program executions** (i.e., for what input values) should one monitor?

Monitoring

- For vulnerabilities concerning **violations of a specified property of a single execution**
  - detection can be performed by **monitoring for violations of that specification**

Monitoring

- For vulnerabilities concerning **violations of a specified property of a single execution**
  - detection can be performed by **monitoring for violations of that specification**
- For other vulnerabilities, or when monitoring for violations of a specification is too expensive, **approximative monitors** can be defined
  - In cases where a dynamic analysis is approximative, it can also generate **false positives** or **false negatives**
    - even though it operates on a concrete execution trace

Monitoring

- For **structured output generation vulnerabilities**, the main **challenge** is:
  - that the intended structure of the generated output is often implicit
    - there exists no explicit specification that can be monitored

Monitoring

- For **structured output generation vulnerabilities**, the main **challenge** is:
  - that the intended structure of the generated output is often implicit
    - there exists no explicit specification that can be monitored
- For example, a monitor can use a **fine-grained dynamic taint analysis** to track the flow of untrusted input strings
  - flag a violation when **untrusted input** has an impact on the parse tree of the generated output

Monitoring

- **Assertions**, **pre-conditions,** and **post-conditions** can be compiled into the code to provide a monitor for API vulnerabilities at testing time
  - even if the cost of these compiled-in run-time checks can be too high to use them in production code

Monitoring

- **Assertions**, **pre-conditions,** and **post-conditions** can be compiled into the code to provide a monitor for API vulnerabilities at testing time
  - even if the cost of these compiled-in run-time checks can be too high to use them in production code
- Monitoring for **race conditions is hard**, but some approaches for monitoring data races on shared memory cells exist
  - E.g., by monitoring whether all shared memory accesses follow a **consistent locking discipline**

LTL – Linear Temporal Logic

Supported operators:

- U: *p* holds **until** *q* holds *p* U *q*

LTL – Linear Temporal Logic

Supported operators:

- U: *p* holds **until** *q* holds   *p* U *q*
- F: *p* will hold eventually in the **future**   F *p*

LTL – Linear Temporal Logic

Supported operators:

• U: *p* holds **until** *q* holds	*p* U *q*
• F: *p* will hold eventually in the **future**	F *p*
• G: *p* **always** holds in the future	G *p*

LTL – Linear Temporal Logic

Supported operators:

- U: *p* holds **until** *q* holds — *p* U *q*
- F: *p* will hold eventually in the **future** — F *p*
- G: *p* **always** holds in the future — G *p*

- X is not well defined for C
  - no notion of “next”

LTL – Linear Temporal Logic

Supported operators:

- U: *p* holds **until** *q* holds — *p* U *q*
- F: *p* will hold eventually in the **future** — F *p*
- G: *p* **always** holds in the future — G *p*

- X is not well defined for C
  - no notion of "next"

- C expressions used as atoms in LTL:

```
{keyInput == 1} -> F {displayKeyUp}

({keyInput != 0} | {intr}) -> G{numInputs > 0}
```

"event": change of global variable used in LTL formula

Büchi Automata (BA)

- non-deterministic FSM over propositional expressions

Büchi Automata (BA)

- non-deterministic FSM over propositional expressions
- inputs infinite length traces

Büchi Automata (BA)

- non-deterministic FSM over propositional expressions
- inputs infinite length traces
- acceptance == trace passes through an accepting state infinitely often

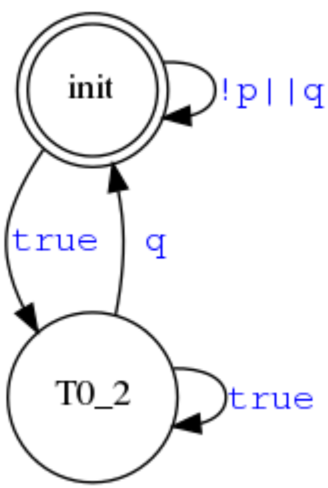
Büchi Automata (BA)

- non-deterministic FSM over propositional expressions
- inputs infinite length traces
- acceptance == trace passes through an accepting state infinitely often
- can convert from LTL to an equivalent BA
  - use ltl2ba, modified to produce C

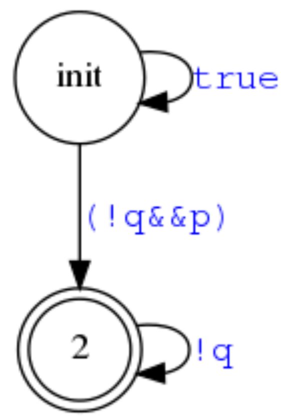
Büchi Automata (BA)

- non-deterministic FSM over propositional expressions
- inputs infinite length traces
- acceptance == trace passes through an accepting state infinitely often

- can convert from LTL to an equivalent BA
  - use ltl2ba, modified to produce C



p -> Fq



!(p -> Fq)

Using BAs to check the program

- Theory: check product of model and *never claim* for accepting state

Using BAs to check the program

- Theory: check product of model and *never claim* for accepting state
- SPIN: execute *never claim* in lockstep with model

Using BAs to check the program

- Theory: check product of model and *never claim* for accepting state
- SPIN: execute *never claim* in lockstep with model
- ESBMC:
  - technically difficult to alternate between normal program and *never claim* program
  - instead: run *never claim* program as a monitor thread concurrently with other program thread(s)

  ⇒ no distinction between monitor thread and other threads

Jeremy Morse, Lucas C. Cordeiro, Denis A. Nicole, Bernd Fischer: Context-Bounded Model Checking of LTL Properties for ANSI-C Software. SEFM 2011: 302-317

Ensuring soundness of monitor thread

Monitor thread will miss events:

- interleavings will exist where events are skipped (monitor thread scheduled out of sync)

⇒ can cause false violations of the property being verified

⇒ monitor thread must be run immediately after events

Ensuring soundness of monitor thread

Monitor thread will miss events:

- interleavings will exist where events are skipped (monitor thread scheduled out of sync)

⇒ can cause false violations of the property being verified

⇒ monitor thread must be run immediately after events

Solution:

- ESBMC maintains (global) current count of events
- monitor checks it processes events one at a time (using assume statements)

⇒ causes ESBMC to discard interleavings where monitor does not act on relevant state changes

Example monitor thread

```
bool cexpr_0;  // “pressed”
bool cexpr_1;  // “charge > min”

typedef enum {T0_init, accept_S2 } ltl2ba_state;
ltl2ba_state state = T0_init;
unsigned int visited_states[2];
unsigned int trans_seen;
extern unsigned int trans_count;

void ltl2ba_fsm(bool state_stats) {
  unsigned int choice;
  while(1) {
    choice = nondet_uint();
    /* Force a context switch */
    yield();
    atomic_begin();
    assume(trans_count <=  trans_seen + 1);
    trans_seen = trans_count;
```

Example monitor thread

```
bool cexpr_0;  // “pressed”
bool cexpr_1;  // “charge > min”

typedef enum {T0_init, accept_S2 } ltl2ba_state;
ltl2ba_state state = T0_init;
unsigned int visited_states[2];
unsigned int trans_seen;
extern unsigned int trans_count;

void ltl2ba_fsm(bool state_stats) {
  unsigned int choice;
  while(1) {
    choice = nondet_uint();
    /* Force a context switch */
    yield();
    atomic_begin();
    assume(trans_count <=  trans_seen + 1);
    trans_seen = trans_count;
```

State transition and “event” counter setup

Example monitor thread

```
bool cexpr_0;  // "pressed"
bool cexpr_1;  // "charge > min"

typedef enum {T0_init, accept_S2 } ltl2ba_state;
ltl2ba_state state = T0_init;
unsigned int visited_states[2];
unsigned int trans_seen;
extern unsigned int trans_count;

void ltl2ba_fsm(bool state_stats) {
  unsigned int choice;
  while(1) {
    choice = nondet_uint();
    /* Force a context switch */
    yield();
    atomic_begin();
    assume(trans_count <=  trans_seen + 1);
    trans_seen = trans_count;
```

State transition and "event" counter setup

nondeterminism

Example monitor thread

```
bool cexpr_0;  // "pressed"
bool cexpr_1;  // "charge > min"

typedef enum {T0_init, accept_S2 } ltl2ba_state;
ltl2ba_state state = T0_init;
unsigned int visited_states[2];
unsigned int trans_seen;
extern unsigned int trans_count;

void ltl2ba_fsm(bool state_stats) {
  unsigned int choice;
  while(1) {
    choice = nondet_uint();
    /* Force a context switch */
    yield();
    atomic_begin();
    assume(trans_count <=  trans_seen + 1);
    trans_seen = trans_count;
```

State transition and "event" counter setup

nondeterminism

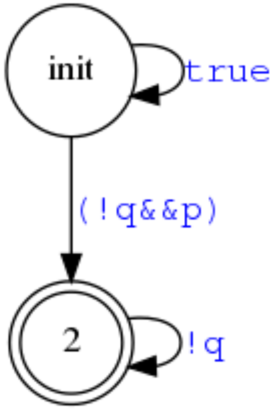
only interleave whole block

reject unsafe interleavings

Example monitor thread

```
    switch(state) {
    case T0_init:
      if(choice == 0) {
        assume((1));
        state = T0_init;
      } else if (choice == 1) {
        assume((!cexpr_1 && cexpr_0));
        state = accept_S2;
      } else assume(0);
      break;
    case accept_S2:
      if(choice == 0) {
        assume((!cexpr_1));
        state = accept_S2;
      } else assume(0);
      break;
    }
    atomic_end();
  }
}
```

automata transitions representing the formula !(p → Fq)



Infinite traces and BMC?

BMC forces program execution to eventually end
- but BA are defined over infinite traces...

Infinite traces and BMC?

BMC forces program execution to eventually end
– but BA are defined over infinite traces...

Solution:

- follow SPINs stuttering acceptance approach:
  pretend final state extends infinitely
- re-run monitor thread after program termination,
  with enough loop iterations to pass through each state twice
- if an accepting state is visited at least twice while stuttering,
  BA accepts extended trace
    - LTL property violation found

Intended learning outcomes

- Understand **dynamic detection techniques** to identify security vulnerabilities
- Generate **executions of the program** along paths that will lead to the **discovery of new vulnerabilities**
- Explain **black-box fuzzing: grammar-based** and **mutation-based fuzzing**
- Explain **white-box fuzzing: dynamic symbolic execution**

Generating relevant executions

Challenge: generate executions of the program along paths that will lead to the discovery of new vulnerabilities

Generating relevant executions

Challenge: generate executions of the program along paths that will lead to the discovery of new vulnerabilities

- This problem is an instance of the general problem in **software testing**
  - Systematically **select appropriate inputs** for a program under test

Generating relevant executions

Challenge: generate executions of the program along paths that will lead to the discovery of new vulnerabilities

- This problem is an instance of the general problem in **software testing**
  - Systematically **select appropriate inputs** for a program under test
  - These techniques are often described by the umbrella term **fuzz testing** or **fuzzing**

Fuzzing

Fuzzing is a highly effective, mostly automated, security testing technique

Fuzzing

Fuzzing is a highly effective, mostly automated, security testing technique

- **Basic idea:** generate random inputs and check whether an application crashes
  - We are not testing functional correctness (compliance)

Fuzzing

Fuzzing is a highly effective, mostly automated, security testing technique

- **Basic idea:** generate random inputs and check whether an application crashes
  - We are not testing functional correctness (compliance)
- **Original fuzzing:** generate long inputs and check whether the system crashes
  - What kind of bug would such a segfault signa*l*?
    - Memory access violation

Fuzzing

Fuzzing is a highly effective, mostly automated, security testing technique

- **Basic idea:** generate random inputs and check whether an application crashes
  - We are not testing functional correctness (compliance)
- **Original fuzzing:** generate long inputs and check whether the system crashes
  - What kind of bug would such a segfault signa*l*?
    - Memory access violation
  - Why would inputs ideally be very long?
    - To make it likely that buffer overruns cross segment boundaries so that the OS triggers a fault

Simple fuzzing ideas

- What inputs would you use for fuzzing?

Simple fuzzing ideas

- What inputs would you use for fuzzing?
  - **very long** or completely **blank strings**

Simple fuzzing ideas

- What inputs would you use for fuzzing?
  - **very long** or completely **blank strings**
  - **min/max** values of integers, or only zero and negative values

Simple fuzzing ideas

- What inputs would you use for fuzzing?
  - **very long** or completely **blank strings**
  - **min/max** values of integers, or only zero and negative values
  - depending on what you are fuzzing, include **unique values**, **characters** or **keywords** likely to trigger bugs:
    - nulls, newlines, or end-of-file characters
    - format string characters **%s %x %n**
    - semi-colons, slashes and backslashes, quotes
    - application-specific keywords **halt, DROP TABLES, …**

Illustrative Example

- Is this circular buffer implementation correct?

```
#define BUFFER_MAX 10
static char buffer[BUFFER_MAX];
int first, next, buffer_size;
void initLog(int max) {
  buffer_size = max;
  first = next = 0;
}
int removeLogElem(void) {
  first++;
  return buffer[first-1];
}
void insertLogElem(int b) {
  if (next < buffer_size) {
    buffer[next] = b;
    next = (next+1)%buffer_size;
  }
}
```

Illustrative Example

- Does this test case expose some error?

```
void testCircularBuffer(void) {
  int senData[] = {1, -128, 98, 88, 59, 1,
-128, 90, 0, -37};
  int i;
  initLog(5);
  for(i=0; i<10; i++)
    insertLogElem(senData[i]);
  for(i=5; i<10; i++)
    assert(senData[i], removeLogElem());
}
```

Illustrative Example

- Does this test case expose some error?

```
void testCircularBuffer(void) {
  int senData[] = {1, -128, 98, 88, 59, 1,
-129, 90, 0, -37};
  int i;
  initLog(5);
  for(i=0; i<10; i++)
    insertLogElem(senData[i]);
  for(i=5; i<10; i++)
    assert(senData[i], removeLogElem());
}
```

Illustrative Example

- Is this circular buffer implementation correct?

```
#define BUFFER_MAX 10
static char buffer[BUFFER_MAX];
int first, next, buffer_size;
void initLog(int max) {
  buffer_size = max;
  first = next = 0;
}
int removeLogElem(void) {
  first++;
  return buffer[first-1];
}
void insertLogElem(int b) {
  if (next < buffer_size) {
    buffer[next] = b;
    next = (next+1)%buffer_size;
  }
}
```

The buffer array is of type char and size BUFFER_MAX

Illustrative Example

- Is this circular buffer implementation correct?

```
#define BUFFER_MAX 10
static char buffer[BUFFER_MAX];
int first, next, buffer_size;
void initLog(int max) {
  buffer_size = max;
  first = next = 0;
}
int removeLogElem(void) {
  first++;
  return buffer[first-1];
}
void insertLogElem(int b) {
  if (next < buffer_size) {
    buffer[next] = b;
    next = (next+1)%buffer_size;
  }
}
```

The buffer array is of type char and size BUFFER_MAX

Increment first without checking the array bound: buffer overflow

Illustrative Example

- Is this circular buffer implementation correct?

```
#define BUFFER_MAX 10
static char buffer[BUFFER_MAX];
int first, next, buffer_size;
void initLog(int max) {
  buffer_size = max;
  first = next = 0;
}
int removeLogElem(void) {
  first++;
  return buffer[first-1];
}
void insertLogElem(int b) {
  if (next < buffer_size) {
    buffer[next] = b;
    next = (next+1)%buffer_size;
  }
}
```

The buffer array is of type char and size BUFFER_MAX

Increment first without checking the array bound: buffer overflow

Assign an integer to a char variable: typecast overflow

Pros & cons of fuzzing

- Minimal effort:
  - the test cases are automatically generated, and test oracle is is merely looking for crashes
- Fuzzing of a C/C++ binary can quickly give a good picture of the robustness of the code

Pros & cons of fuzzing

- Minimal effort:
  - the test cases are automatically generated, and test oracle is is merely looking for crashes
- Fuzzing of a C/C++ binary can quickly give a good picture of the robustness of the code

- Fuzzers do not find all bugs
- Crashes may be hard to analyze, but a crash is a **true positive** that something is wrong!
- For programs that take **complex inputs**, more work will be needed to get **reasonable code coverage** and **hit unusual test cases**
  - Leads to various studies on **"smarter" fuzzers**

Intended learning outcomes

- Understand **dynamic detection techniques** to identify security vulnerabilities

- Generate **executions of the program** along paths that will lead to the **discovery of new vulnerabilities**

- Explain **black-box fuzzing: grammar-based** and **mutation-based fuzzing**

- Explain **white-box fuzzing: dynamic symbolic execution**

Black-box fuzzing

The generation of values depends on the program input/output behaviour, and not on its internal structure

Black-box fuzzing

The generation of values depends on the program input/output behaviour, and not on its internal structure

① **Random testing:** input values are randomly sampled from the appropriate value domain

Black-box fuzzing

The generation of values depends on the program input/output behaviour, and not on its internal structure

① **Random testing:** input values are randomly sampled from the appropriate value domain

② **Grammar-based fuzzing:** a model of the expected format of input values is taken into account during the generation of input values

Black-box fuzzing

The generation of values depends on the program input/output behaviour, and not on its internal structure

① **Random testing:** input values are randomly sampled from the appropriate value domain

② **Grammar-based fuzzing:** a model of the expected format of input values is taken into account during the generation of input values

③ **Mutation-based fuzzing:** the fuzzer is provided with typical input values; it generates new input values by performing small mutations on the provided input

Random Testing

- Random testing produces **random, independent inputs**, to test software

```
int sig_invert(int signal) {
  if (signal < 0)
    return signal; // bug
  else
    return signal;
}
```

Random Testing

- Random testing produces **random, independent inputs**, to test software

```
int sig_invert(int signal) {
  if (signal < 0)
    return signal; // bug
  else
    return signal;
}
```

```
void testSig_Inverter(int n) {
    for (int i=0; i<n; i++) {
        int x = rand();
        int result = sig_invert(x);
        assert(result >= 0);
    }
}
```

Random Testing

- Random testing produces **random, independent inputs**, to test software

```
int sig_invert(int signal) {
  if (signal < 0)
    return signal; // bug
  else
    return signal;
}
```

the random tests could be {827989654, 328082218, 1487316077, 611655059, 82358424}

```
void testSig_Inverter(int n) {
    for (int i=0; i<n; i++) {
        int x = rand();
        int result = sig_invert(x);
        assert(result >= 0);
    }
}
```

Replace random by non-deterministic variable

- Use a model checker to produce an **input that triggers the property violation**

```
int nondet_int();
void testSig_Inverter(int n) {
  for (int i=0; i<n; i++) {
    int x = nondet_int (); //rand();
    int result = sig_invert(x);
    assert(result >= 0);
  }
}
```

Replace random by non-deterministic variable

- Use a model checker to produce an **input that triggers the property violation**

```
int nondet_int();
void testSig_Inverter(int n) {
  for (int i=0; i<n; i++) {
    int x = nondet_int (); //rand();
    int result = sig_invert(x);
    assert(result >= 0);
  }
}
```

Replace random by non-deterministic variable

- Use a model checker to produce an **input that triggers the property violation**

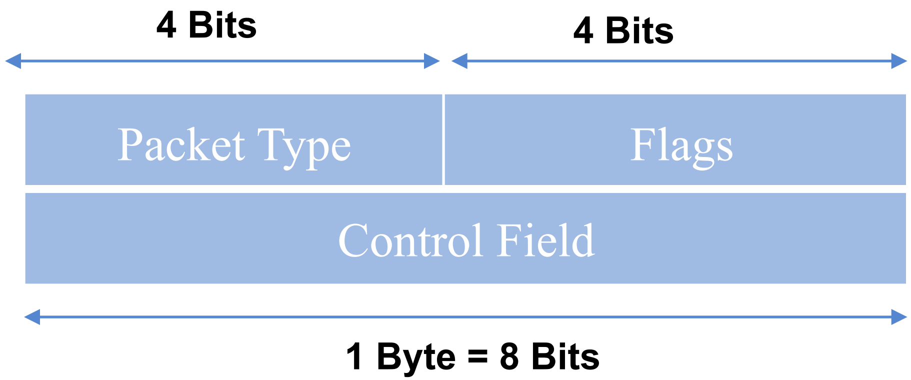
```
int nondet_int();
void testSig_Inverter(int n) {
  for (int i=0; i<n; i++) {
    int x = nondet_int (); //rand();
    int result = sig_invert(x);
    assert(result >= 0);
  }
}
```

$esbmc random-testing.c

```
State 9 file file.c line 16 function testSig_Inverter thread 0
----------------------------------------------------
  x = -2147483648
…
Violated property:
…
!((_Bool)((signed long int)(!(result >= 0))))
```

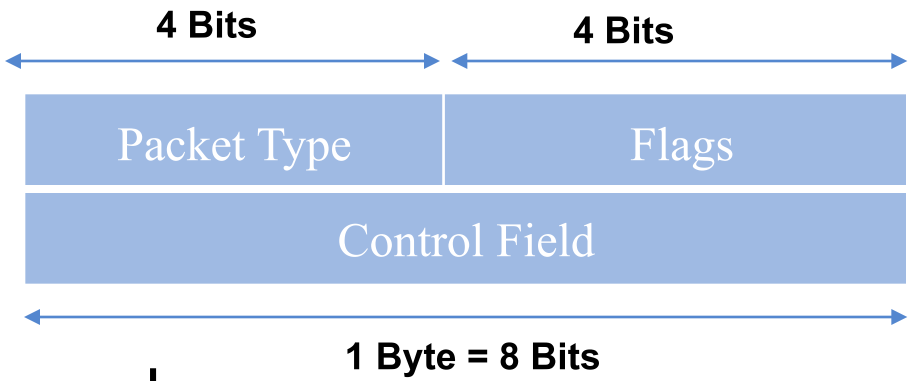
Grammar-based fuzzing

- For communication protocols, a **grammar-based fuzzer** generate files or data packets, which are:
  - Slightly malformed
  - Hit corner cases in the spec
  - **Grammar** defining legal input or **a data format specification**



Grammar-based fuzzing

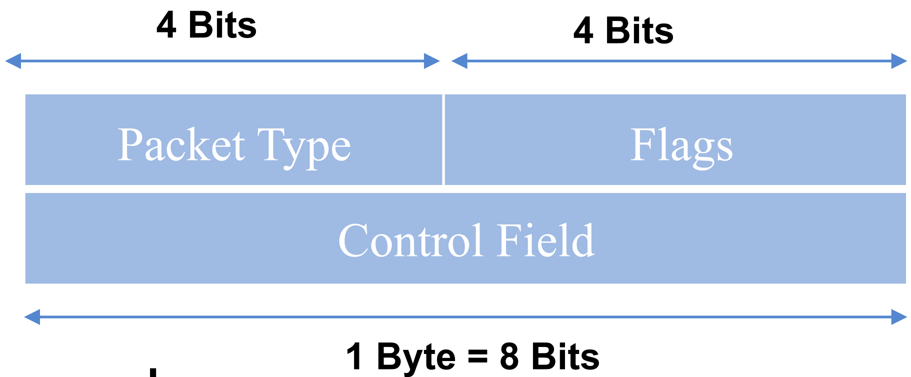
- For communication protocols, a **grammar-based fuzzer** generate files or data packets, which are:
  - Slightly malformed
  - Hit corner cases in the spec
  - **Grammar** defining legal input or **a data format specification**



- Typical things that can be fuzzed:
  - many/all possible value for specific fields (undefined values)
  - incorrect lengths, lengths that are zero, or payloads that are too short/long

Grammar-based fuzzing

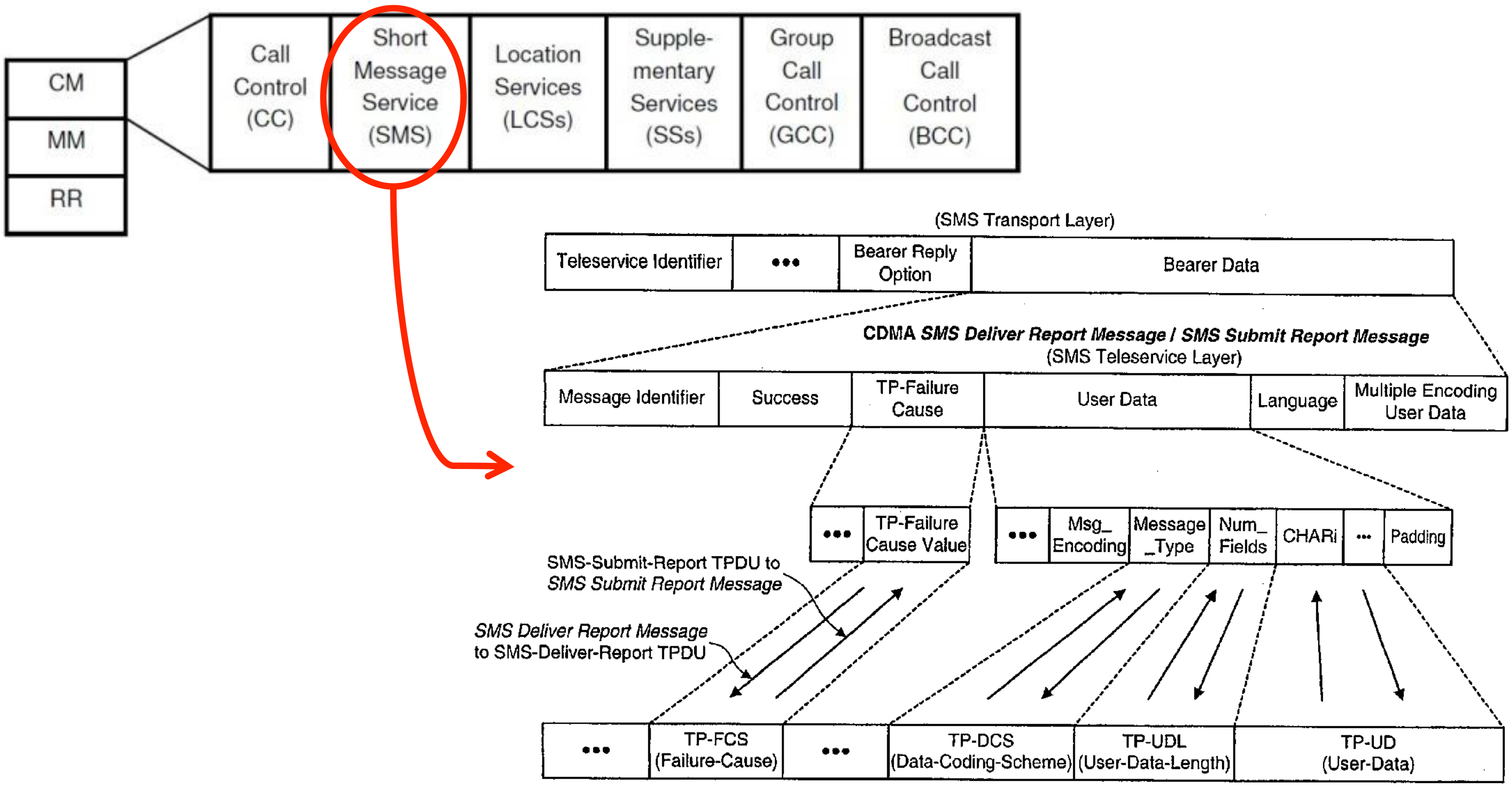
- For communication protocols, a **grammar-based fuzzer** generate files or data packets, which are:
  - Slightly malformed
  - Hit corner cases in the spec
  - **Grammar** defining legal input or **a data format specification**



- Typical things that can be fuzzed:
  - many/all possible value for specific fields (undefined values)
  - incorrect lengths, lengths that are zero, or payloads that are too short/long
- Tools for building such fuzzers: SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, ...

Example: Grammar-based Fuzzing of GSM

GSM is an extremely rich and complicated protocol



Fabian van den Broek, Brinio Hond, Arturo Cedillo Torres: Security Testing of GSM Implementations. ESSoS 2014: 179-195

SMS Message Fields

Field	size
Message Type Indicator	2 bit
Reject Duplicates	1 bit
Validity Period Format	2 bit
User Data Header Indicator	1 bit
Reply Path	1 bit
Message Reference	integer
Destination Address	2-12 byte
Protocol Identifier	1 byte
Data Coding Scheme (CDS)	1 byte
Validity Period	1 byte/7 bytes
User Data Length (UDL)	integer
User Data	depends on CDS and UDL

Example: GSM protocol fuzzing

- We can use a **Universal Software Radio Peripheral** (USRP)
  - Most USRPs connect to a host computer through a high-speed link
    - the host-based software uses to control the USRP hardware and transmit/receive data
  - With open-source cell tower software (OpenBTS) to fuzz any phone

Example: GSM protocol fuzzing

- Fuzzing SMS layer of GSM reveals unexpected behaviour in GSM standard and phones

Example: GSM protocol fuzzing

- Fuzzing SMS layer of GSM reveals unexpected behaviour in GSM standard and phones

you have a fax!

possibility to receive faxes?

Only way to get rid if this icon; reboot the phone

Example: GSM protocol fuzzing

- Malformed SMS text messages
  - show **raw memory** instead of the **text message**

(a) Showing garbage

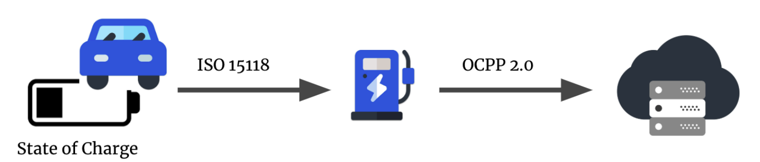
(b) Showing the name of a wallpaper and two games

Mutation-based fuzzing: Fuzzing OCPP

- The **Open Charge Point Protocol** (OCPP) is an application protocol
  - communication between Electric vehicle (EV) charging stations and a central management system
- OCPP can use XML or JSN messages

Example message in JSN format

```
{ "location": NijmegenMercator2156
  "retries": 5,
  "retryInterval": 30,
  "startTime": "2018-10-27T19:10:11",
  "stopTime":  "2018-10-27T22:10:11"  }
```



Mutation-based fuzzing: Fuzzing OCPP

- Simple classification of messages into
  ① **malformed JSN/XML:** missing quote, bracket or comma

Mutation-based fuzzing: Fuzzing OCPP

- Simple classification of messages into
  ① **malformed JSN/XML:** missing quote, bracket or comma
  ② **well-formed JSN/XML, but not legal OCPP:** use field names that are not in the OCPP specs

Mutation-based fuzzing: Fuzzing OCPP

- Simple classification of messages into
  ① **malformed JSN/XML:** missing quote, bracket or comma
  ② **well-formed JSN/XML, but not legal OCPP:** use field names that are not in the OCPP specs
  ③ **well-formed OCPP:** can be used for a simple test oracle
    - Malformed messages (type 1 & 2) should generate a generic error response
    - Well-formed messages (type 3) should not
    - The application should never crash

Mutation-based fuzzing: Fuzzing OCPP

- Simple classification of messages into
  ① **malformed JSN/XML:** missing quote, bracket or comma
  ② **well-formed JSN/XML, but not legal OCPP:** use field names that are not in the OCPP specs
  ③ **well-formed OCPP:** can be used for a simple test oracle
    - Malformed messages (type 1 & 2) should generate a generic error response
    - Well-formed messages (type 3) should not
    - The application should never crash
- Note: this does not require any understanding of the protocol semantics yet!
  – Figuring out correct responses to type 3 would need

Evolutionary Fuzzing with AFL

- **Grammar-based fuzzer:**
  - Significant work to write code to fuzz, even if we use tools to generate this code based on some grammar
- **Mutation-based fuzzer:**
  - The chance that random changes in inputs hit unusual cases is small

Evolutionary Fuzzing with AFL

- **Grammar-based fuzzer:**
  - Significant work to write code to fuzz, even if we use tools to generate this code based on some grammar
- **Mutation-based fuzzer:**
  - The chance that random changes in inputs hit unusual cases is small

- **AFL** (American Fuzzy Lop) takes an evolutionary approach to learn mutations based on measuring code coverage
  - basic idea: if a mutation of the input triggers a new path through the code, then it is an exciting mutation; otherwise, the mutation is discarded
  - Produce random mutations of the input and observe their effect on code coverage, **AFL** can learn what interesting inputs are

The Fuzzing Process of AFL

1. Start with sample seed inputs
2. Mutate seed inputs to generate mutants
3. Collect code coverage (CFG edges) information
4. Save as new seeds if coverage increases
5. Repeat from step 2

/lcamtuf.coredump.cx/afl/https:/

American Fuzzy Lop

- Support programs written in **C/C++/Objective C** and variants for **Python/Go/Rust/OCaml**

https://lcamtuf.coredump.cx/afl/

American Fuzzy Lop

- Support programs written in **C/C++/Objective C** and variants for **Python/Go/Rust/OCaml**
- Code instrumented to observe execution paths:
  - if source code is available, then use **modified compiler**; otherwise, **run code in an emulator**

https://lcamtuf.coredump.cx/afl/

American Fuzzy Lop

- Support programs written in **C/C++/Objective C** and variants for **Python/Go/Rust/OCaml**
- Code instrumented to observe execution paths:
  - if source code is available, then use **modified compiler**; otherwise, **run code in an emulator**
- Code coverage represented as a 64KB bitmap, where control flow jumps are mapped to changes in this bitmap
  - different executions could lead to the same bitmap, but the chance is small

https://lcamtuf.coredump.cx/afl/

American Fuzzy Lop

- Support programs written in **C/C++/Objective C** and variants for **Python/Go/Rust/OCaml**
- Code instrumented to observe execution paths:
  - if source code is available, then use **modified compiler**; otherwise, **run code in an emulator**
- Code coverage represented as a 64KB bitmap, where control flow jumps are mapped to changes in this bitmap
  - different executions could lead to the same bitmap, but the chance is small
- Mutation strategies: bit flips, incrementing/decrementing integers, using pre-defined integer values (e.g., 0, -1, MAX_INT,....), deleting/combining/zeroing input blocks

https://lcamtuf.coredump.cx/afl/

AFL's instrumentation of compiled code

- Code is injected at every branch point in the code

```
cur_location = <COMPILE_TIME_RANDOM_FOR_THIS_CODE_BLOCK>;
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
```

Bitwise exclusive OR

where **shared_mem** is a 64 KB memory region

```
cur_location = 5;
prev_location = 3;
```

0101 (decimal 5)
XOR 0011 (decimal 3)
= 0110 (decimal 6)

cur_location ^ prev_location

AFL's instrumentation of compiled code

- Code is injected at every branch point in the code

```
cur_location = <COMPILE_TIME_RANDOM_FOR_THIS_CODE_BLOCK>;
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
```

Shift right

where **shared_mem** is a 64 KB memory region

```
prev_location = cur_location >> 1;
```

0101 (decimal 5)
shift 0010 (decimal 2)

AFL's instrumentation of compiled code

- Code is injected at every branch point in the code

```
cur_location = <COMPILE_TIME_RANDOM_FOR_THIS_CODE_BLOCK>;
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
```

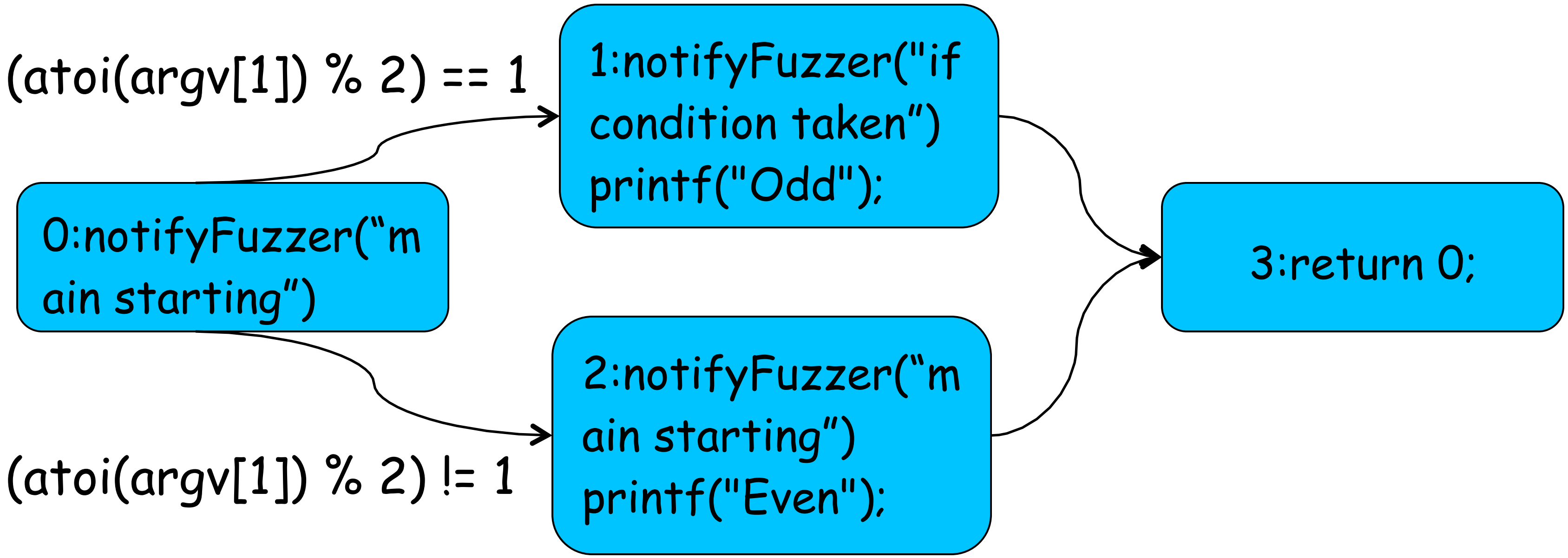
where **shared_mem** is a 64 KB memory region

- Intuition: for every jump from **src** to **dest** in the code a different byte in **shared_mem** is changed
  - This byte is determined by the compile-time randoms inserted at source and destination

Example of AFL instrumentation

- Consider a code fragment that determines a parameter to be **even** or **odd**

```
#include <stdio.h>
#include <stdlib.h>
int main(int arc, char *argv[]) {
  ((atoi(argv[1]) % 2) == 1) ? printf("Odd") : printf("Even");
  return 0;
}
```

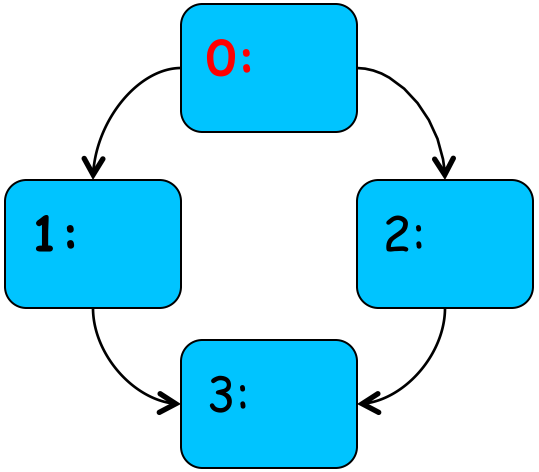


Example of AFL instrumentation

- AFL assigns a **random compile time** constant to **each basic block** and uses a 64kB array to **trace the execution flow** using the following logic

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

```
prev_location: 0
cur_location: 0
cur_location ^ prev_location: 0
shared_mem[0]: 1
prev_location: 0
```

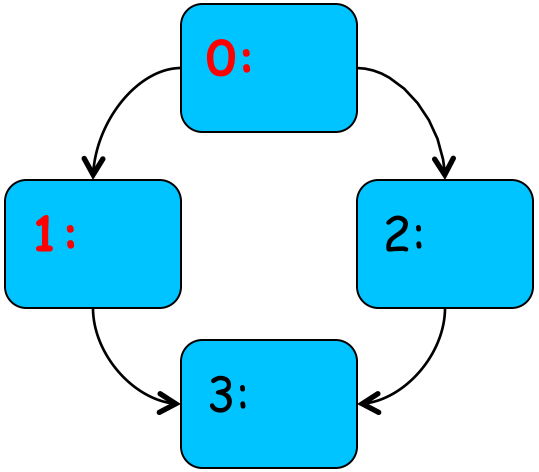


Example of AFL instrumentation

- AFL assigns a **random compile time** constant to **each basic block** and uses a 64kB array to **trace the execution flow** using the following logic

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

```
prev_location: 0
cur_location: 1
cur_location ^ prev_location: 1
shared_mem[1]: 1
prev_location: 0
```

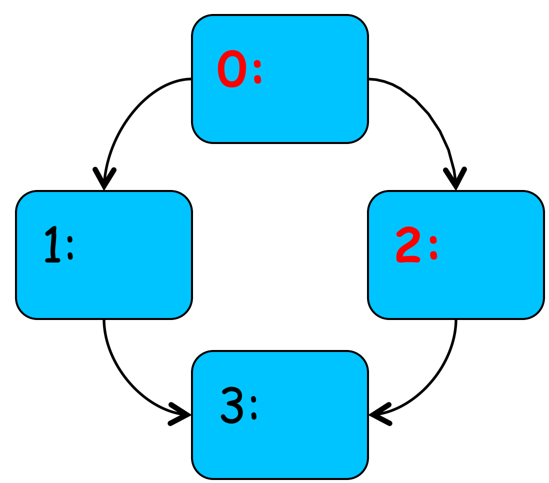


Example of AFL instrumentation

- AFL assigns a **random compile time** constant to **each basic block** and uses a 64kB array to **trace the execution flow** using the following logic

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

```
prev_location: 0
cur_location: 2
cur_location ^ prev_location: 2
shared_mem[2]: 1
prev_location: 1
```

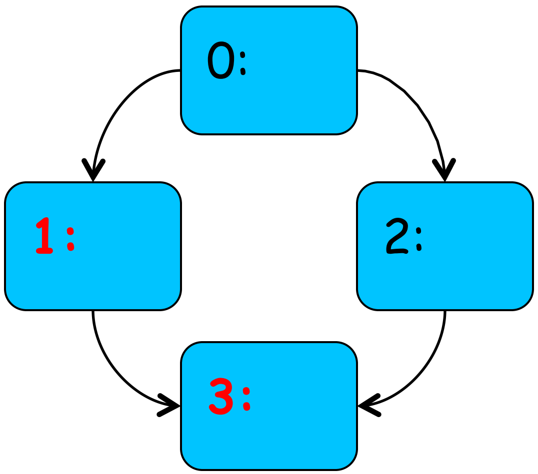


Example of AFL instrumentation

- AFL assigns a **random compile time** constant to **each basic block** and uses a 64kB array to **trace the execution flow** using the following logic

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

```
prev_location: 1
cur_location: 3
cur_location ^ prev_location: 2
shared_mem[2]: 2
prev_location: 1
```

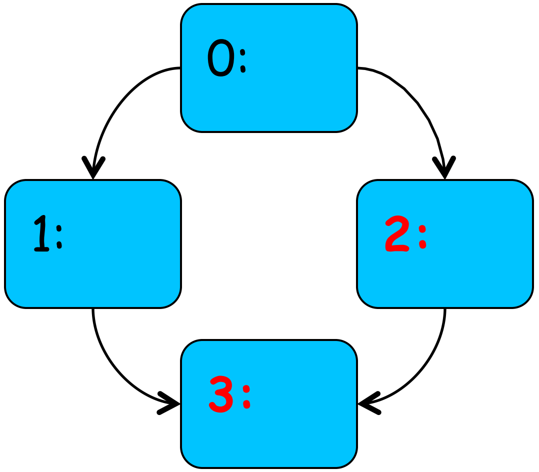


Example of AFL instrumentation

- AFL assigns a **random compile time** constant to **each basic block** and uses a 64kB array to **trace the execution flow** using the following logic

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

```
prev_location: 2
cur_location: 3
cur_location ^ prev_location: 1
shared_mem[1]: 2
prev_location: 1
```



Intended learning outcomes

- Understand **dynamic detection techniques** to identify security vulnerabilities
- Generate **executions of the program** along paths that will lead to the **discovery of new vulnerabilities**
- Explain **black-box fuzzing: grammar-based** and **mutation-based fuzzing**
- Explain **white-box fuzzing: dynamic symbolic execution**

White-box fuzzing

The internal structure of the program is analysed to assist in the generation of appropriate input values

White-box fuzzing

The internal structure of the program is analysed to assist in the generation of appropriate input values

- The primary systematic white-box fuzzing technique is a **dynamic symbolic execution**
  - Executes a program with concrete input values and builds at the same time a **path condition**
    - An expression that specifies the constraints on those input values that have to be fulfilled to take this specific execution path

White-box fuzzing

The internal structure of the program is analysed to assist in the generation of appropriate input values

- The primary systematic white-box fuzzing technique is a **dynamic symbolic execution**
  - Executes a program with concrete input values and builds at the same time a **path condition**
    - An expression that specifies the constraints on those input values that have to be fulfilled to take this specific execution path
  - Solve input values that do not satisfy the path condition of the current execution
    - the fuzzer can make sure that these input values will drive the program to a different execution path, thus **improving coverage**

Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

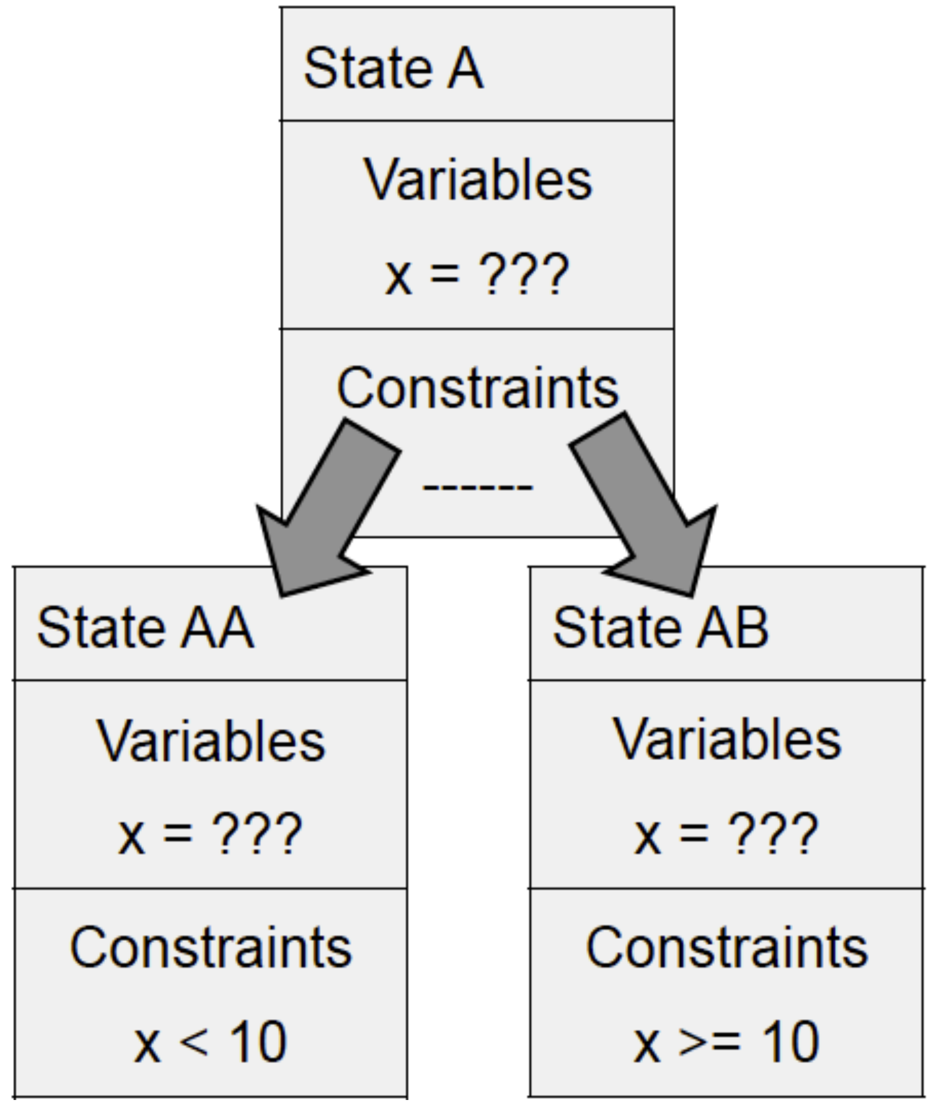
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

State A
Variables
x = ???
Constraints

Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



Coverage Test Generation for Security

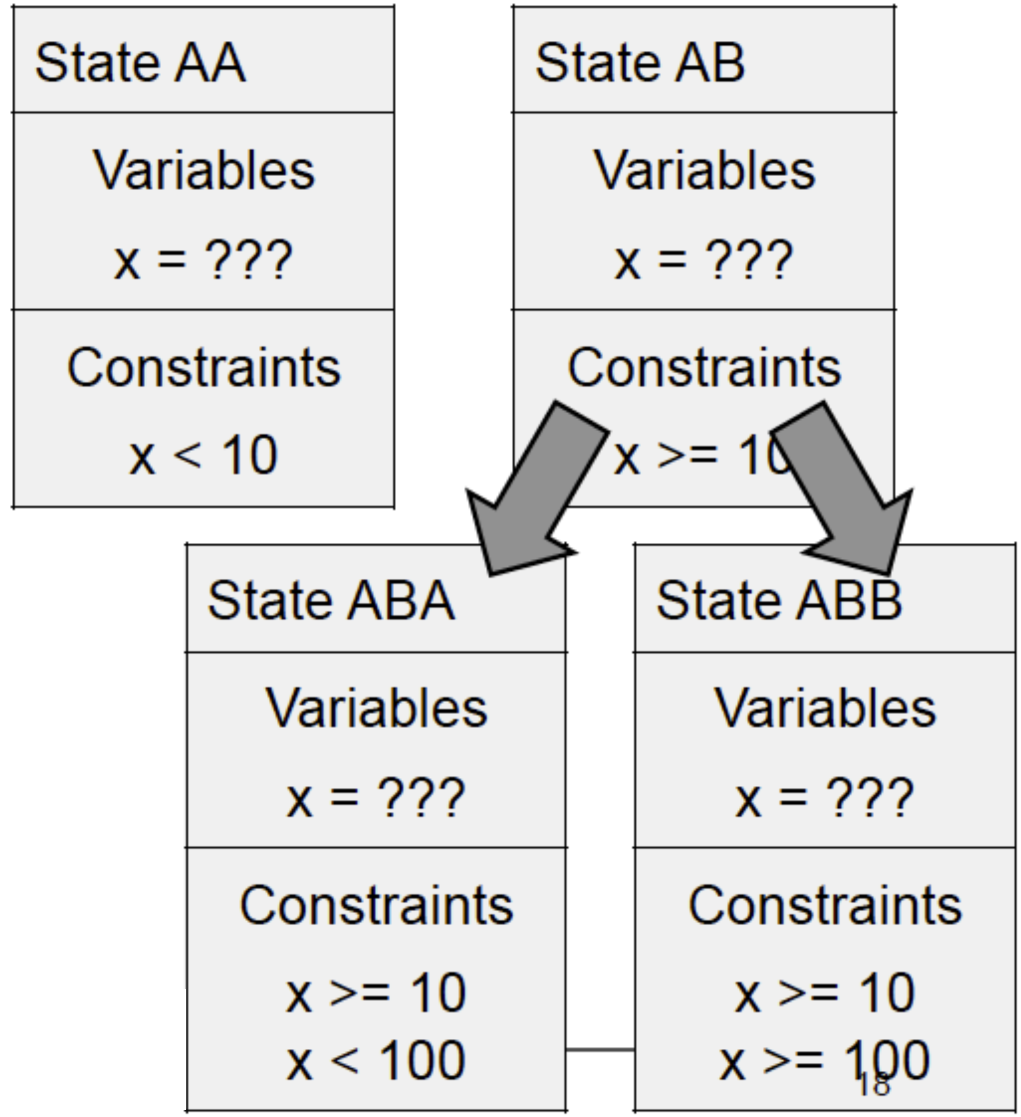
```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

State AA
Variables x = ???
Constraints x < 10

State AB
Variables x = ???
Constraints x >= 10

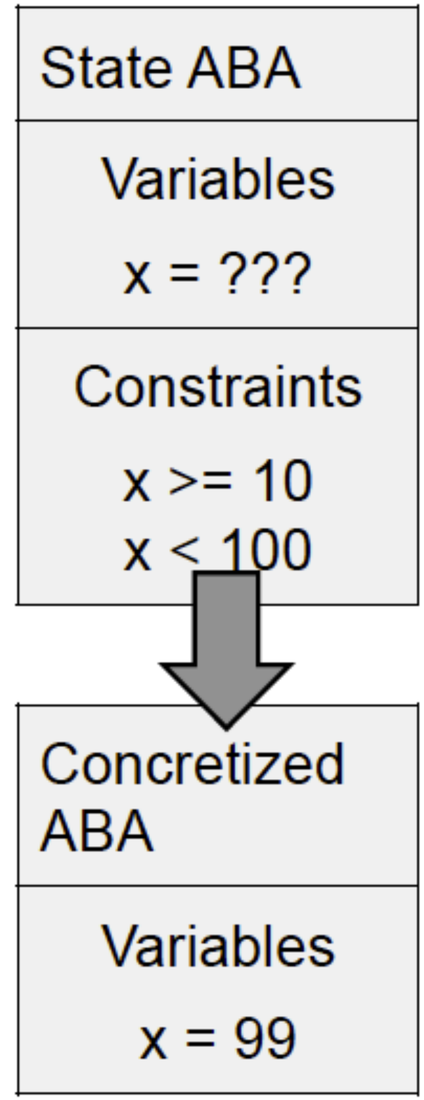
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



White-box Fuzzing

- Combine fuzz testing with **dynamic test generation**

White-box Fuzzing

- Combine fuzz testing with **dynamic test generation**
  - **Run the code** with some initial input

White-box Fuzzing

- Combine fuzz testing with **dynamic test generation**
  - **Run the code** with some initial input
  - **Collect constraints on input** with symbolic execution

White-box Fuzzing

- Combine fuzz testing with **dynamic test generation**
  - **Run the code** with some initial input
  - **Collect constraints on input** with symbolic execution
  - **Generate new constraints**

White-box Fuzzing

- Combine fuzz testing with **dynamic test generation**
  - **Run the code** with some initial input
  - **Collect constraints on input** with symbolic execution
  - **Generate new constraints**
  - **Solve constraints** with constraint solver

White-box Fuzzing

- Combine fuzz testing with **dynamic test generation**
  - **Run the code** with some initial input
  - **Collect constraints on input** with symbolic execution
  - **Generate new constraints**
  - **Solve constraints** with constraint solver
  - **Synthesize new inputs**

White-box Fuzzing

- Combine fuzz testing with **dynamic test generation**
  - **Run the code** with some initial input
  - **Collect constraints on input** with symbolic execution
  - **Generate new constraints**
  - **Solve constraints** with constraint solver
  - **Synthesize new inputs**
  - Leverages **Directed Automated Random Testing** (**DART**) ([Godefroid-Klarlund-Sen-05,…])
  - See also previous talk on **EXE** [Cadar-Engler-05, Cadar-Ganesh-Pawlowski-Engler-Dill-06, Dunbar-Cadar-Pawlowski-Engler-08,…]

Dynamic Test Generation

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 3) crash();
}
```

```
input =
   "good"
```

Dynamic Test Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] ==  'b' ) cnt++;
    if (input[1] ==  'a' ) cnt++;
    if (input[2] ==  'd' ) cnt++;
    if (input[3] ==  '!' ) cnt++;
    if (cnt >= 3) crash();
}
```

input = "good"

I_0 != 'b'

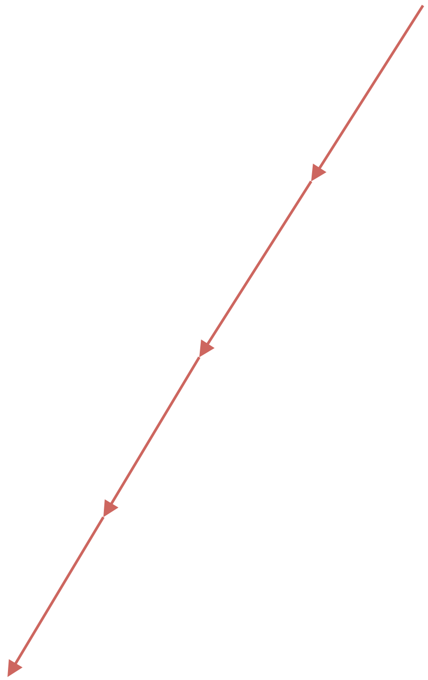
I_1 != 'a'

I_2 != 'd'

I_3 != '!'

Collect constraints from trace
Create new constraints
Solve new constraints → new input.

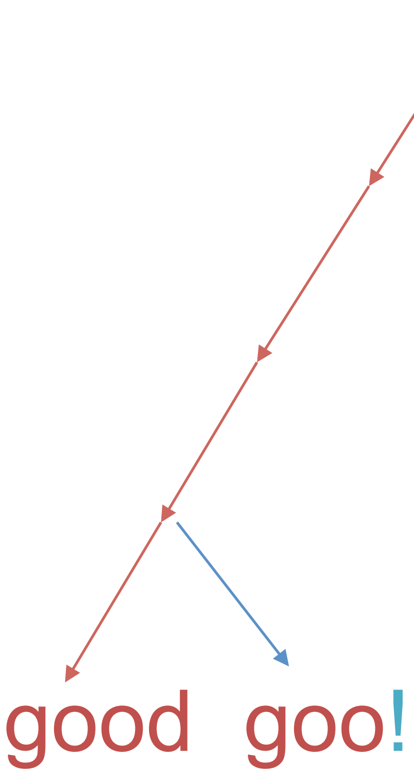
Depth-First Search



good

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] ==  'b' ) cnt++;   I0 != 'b'
    if (input[1] ==  'a' ) cnt++;   I1 != 'a'
    if (input[2] ==  'd' ) cnt++;   I2 != 'd'
    if (input[3] ==  '!' ) cnt++;   I3 != '!'
    if (cnt >= 3) crash();
}
```

Depth-First Search




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] ==  'b' ) cnt++;
    if (input[1] ==  'a' ) cnt++;
    if (input[2] ==  'd' ) cnt++;
    if (input[3] ==  '!' ) cnt++;
    if (cnt >= 3) crash();
}
```

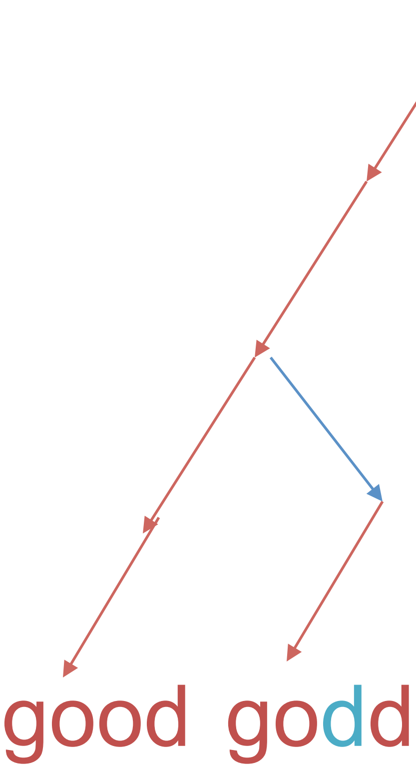
I_0 != 'b'

I_1 != 'a'

I_2 != 'd'

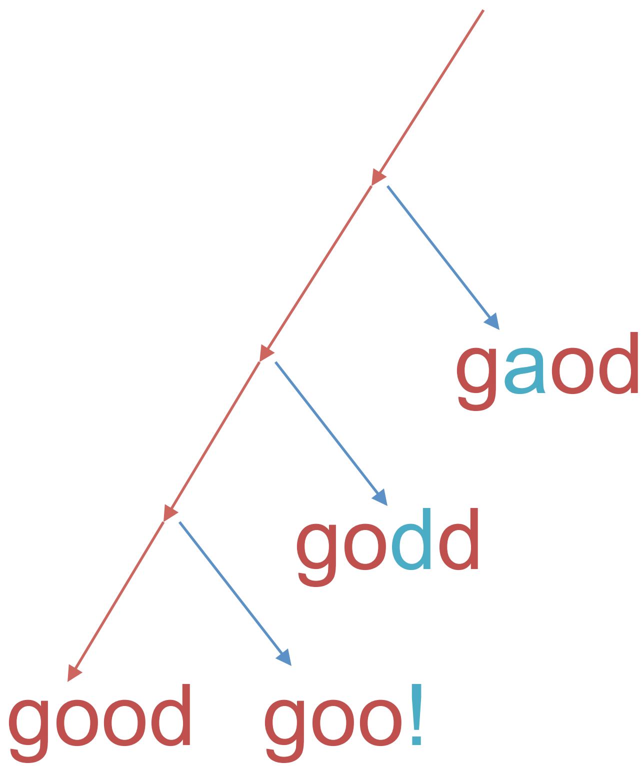
I_3 == '!'

Depth-First Search



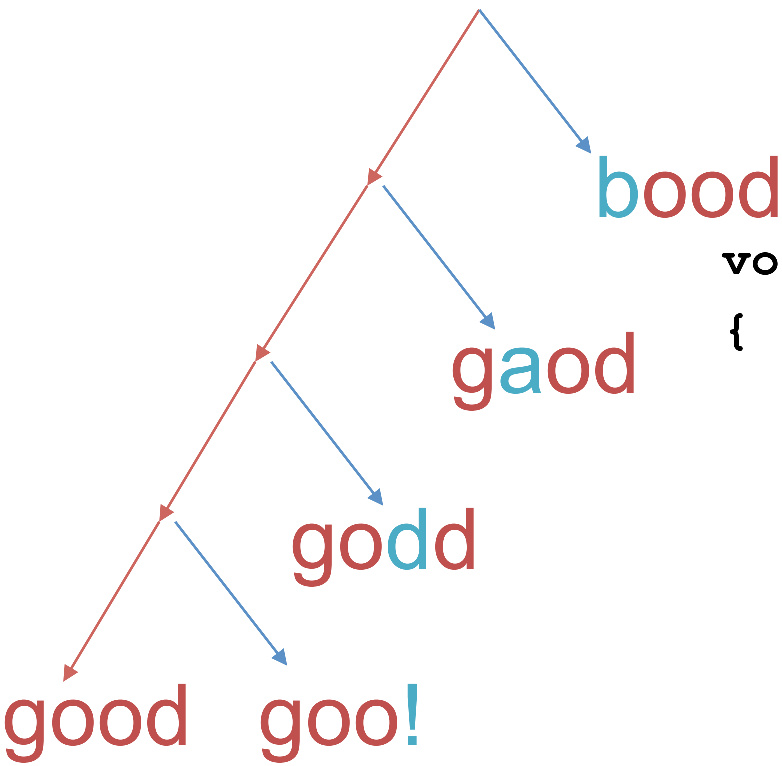

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] ==  'b' ) cnt++; I0 != 'b'
    if (input[1] ==  'a' ) cnt++; I1 != 'a'
    if (input[2] ==  'd' ) cnt++; I2 == 'd'
    if (input[3] ==  '!' ) cnt++; I3 != '!'
    if (cnt >= 3) crash();
}
```

Depth-First Search




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] ==  'b' ) cnt++;  I0 != 'b'
    if (input[1] ==  'a' ) cnt++;  I1 == 'a'
    if (input[2] ==  'd' ) cnt++;  I2 != 'd'
    if (input[3] ==  '!' ) cnt++;  I3 != '!'
    if (cnt >= 3) crash();
}
```

Depth-First Search




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] ==  'b' ) cnt++;
    if (input[1] ==  'a' ) cnt++;
    if (input[2] ==  'd' ) cnt++;
    if (input[3] ==  '!' ) cnt++;
    if (cnt >= 3) crash();
}
```

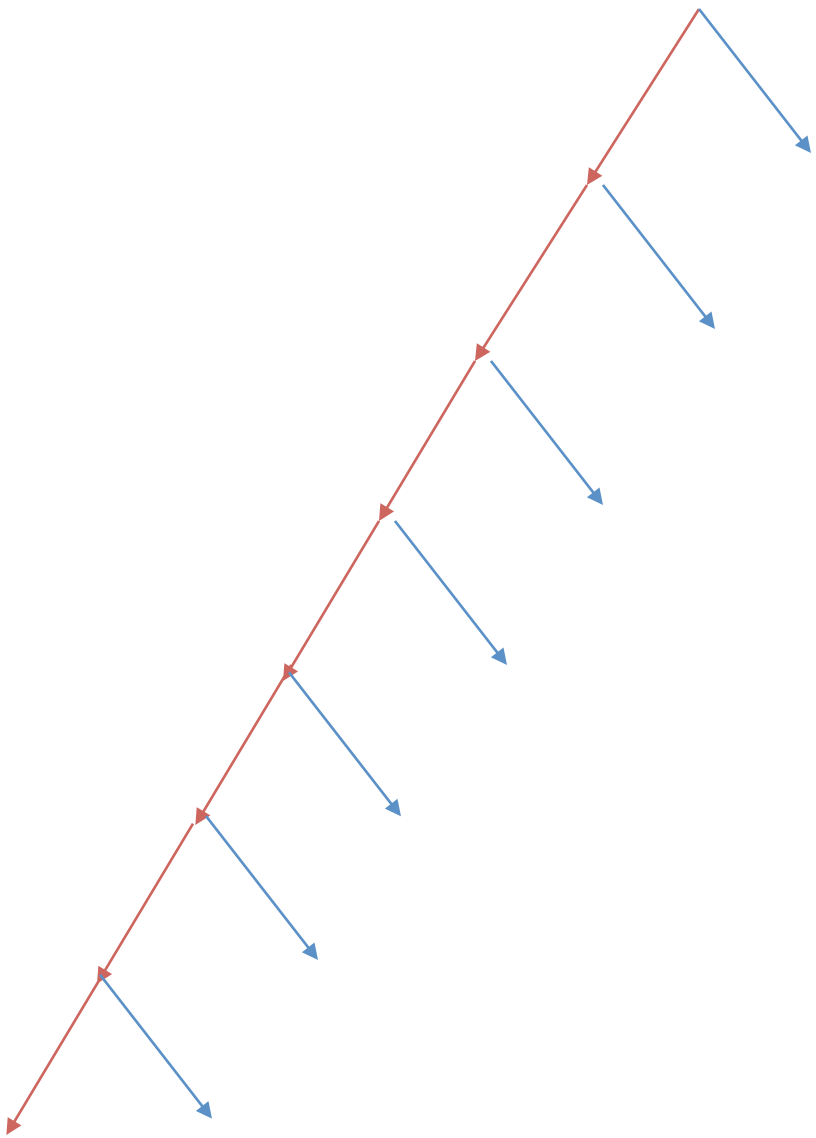
I_0 == 'b'

I_1 != 'a'

I_2 != 'd'

I_3 != '!'

Key Idea: One Trace, Many Tests

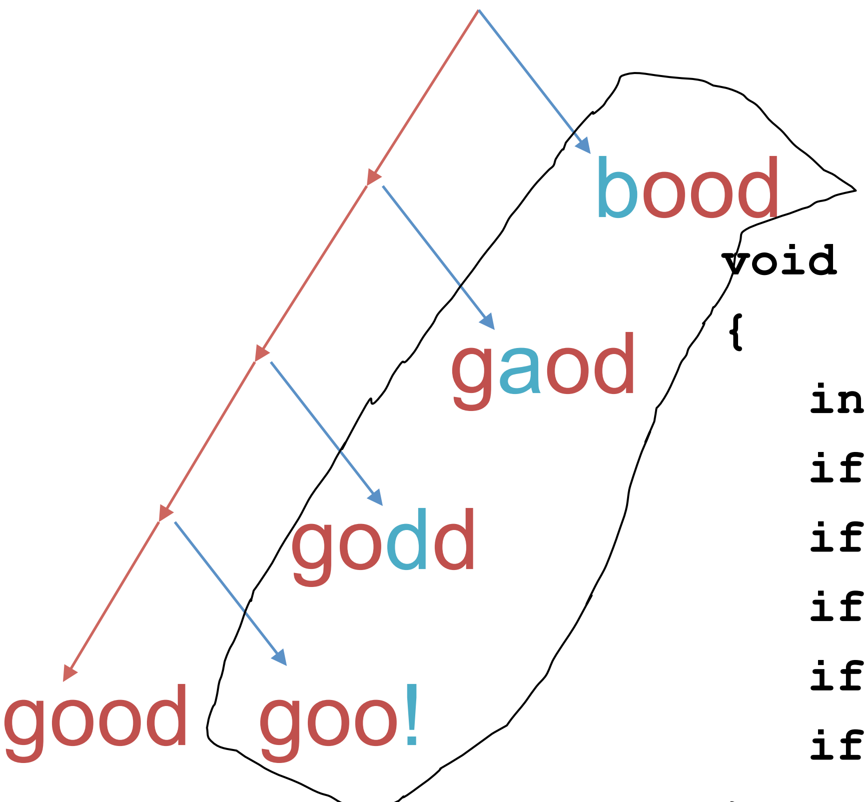


Office 2007 application:
Time to **gather constraints**: 25m30s
Tainted branches/trace: ~1000

Time per branch to
solve,
generate new test,
check for crashes: ~1s

Therefore, solve+check **all** branches for each trace!

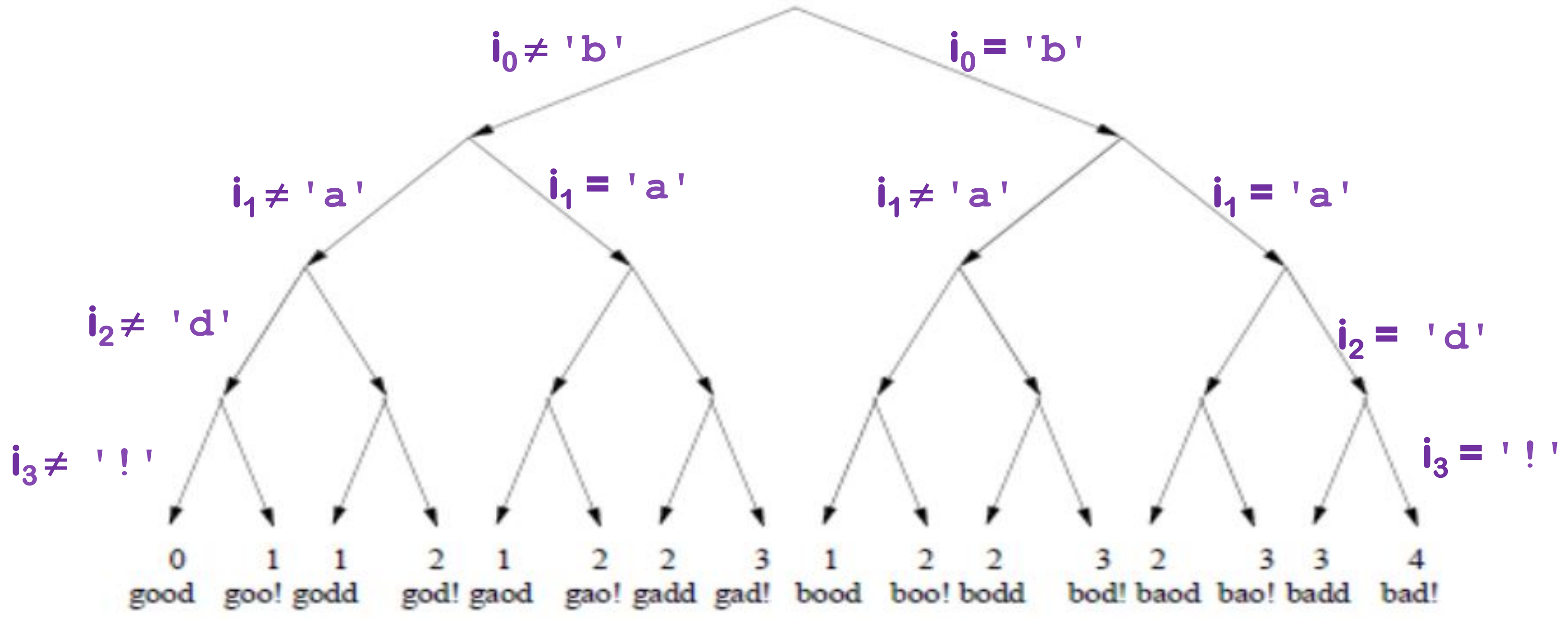
Generational Search




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] ==  'b' ) cnt++;  I0 == 'b'
    if (input[1] ==  'a' ) cnt++;  I1 == 'a'
    if (input[2] ==  'd' ) cnt++;  I2 == 'd'
    if (input[3] ==  '!' ) cnt++;  I3 == '!'
    if (cnt >= 3) crash();
}
```

Search space for interesting inputs

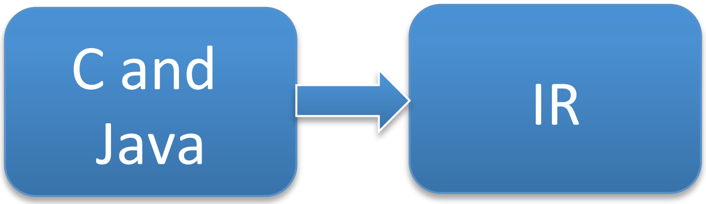
Based on this one execution, combining all these constraints now yields 16 test cases



Note: the initial execution with the input '**good**' was not very interesting, but these others are

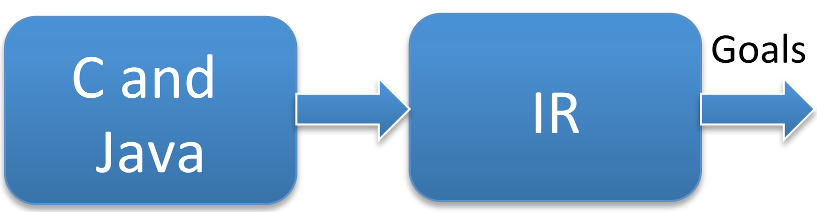
BMC for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)



BMC for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)
- Add goals indicating the **coverage**
  - *location, branch, decision, condition and path*

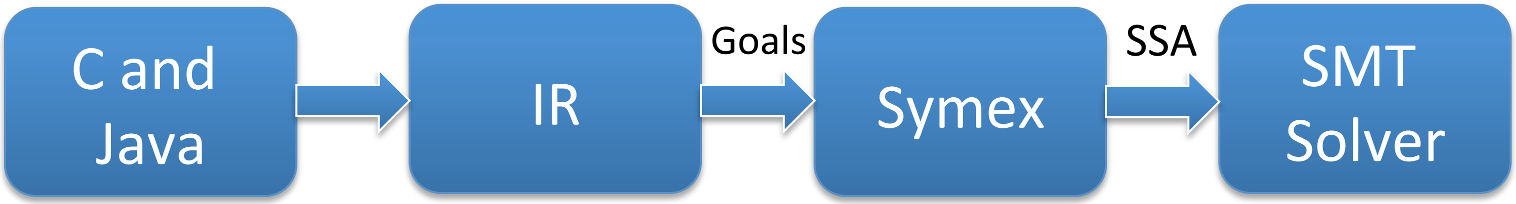


BMC for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)
- Add goals indicating the **coverage**
  - *location, branch, decision, condition and path*
- **Symbolically** execute IR to produce an SSA program

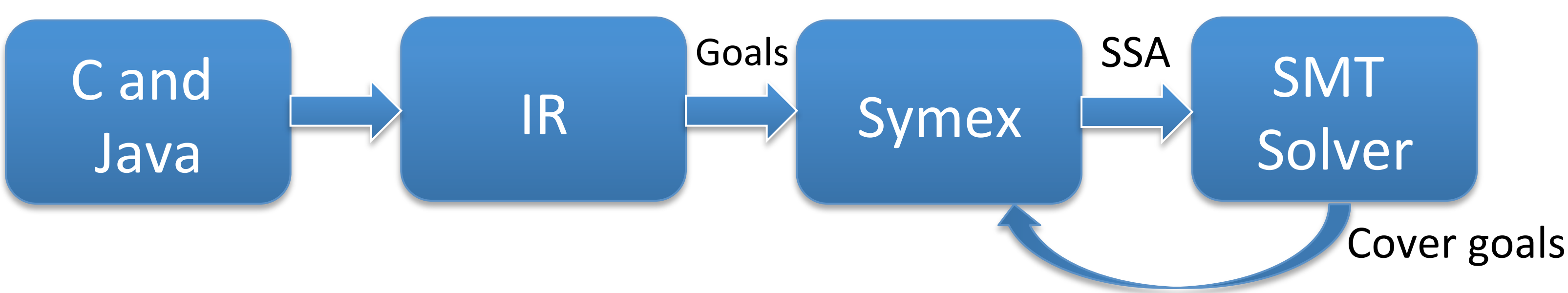
BMC for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)
- Add goals indicating the **coverage**
  - *location, branch, decision, condition and path*
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**



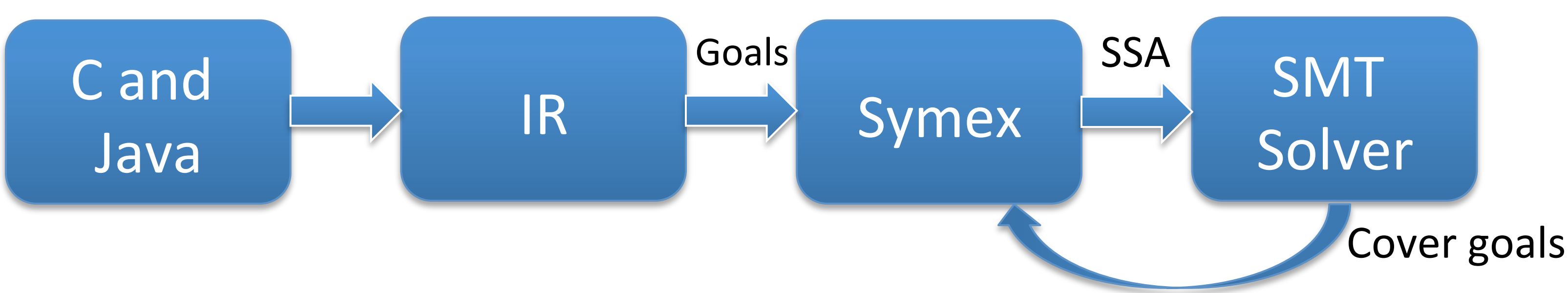
BMC for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)
- Add goals indicating the **coverage**
  - *location, branch, decision, condition and path*
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**
- Solve the formula iteratively to cover different goals



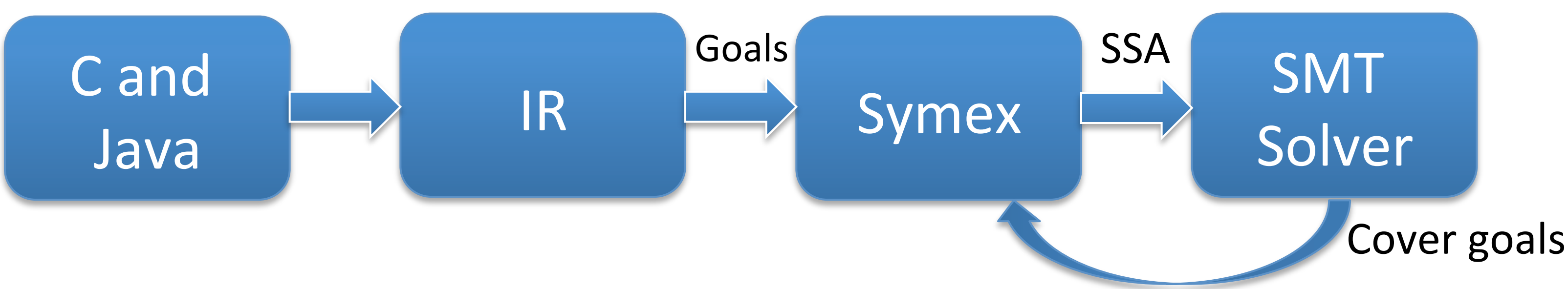
BMC for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)
- Add goals indicating the **coverage**
  - *location, branch, decision, condition and path*
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**
- Solve the formula iteratively to cover different goals
- Interpret the solution to figure out the **input conditions**

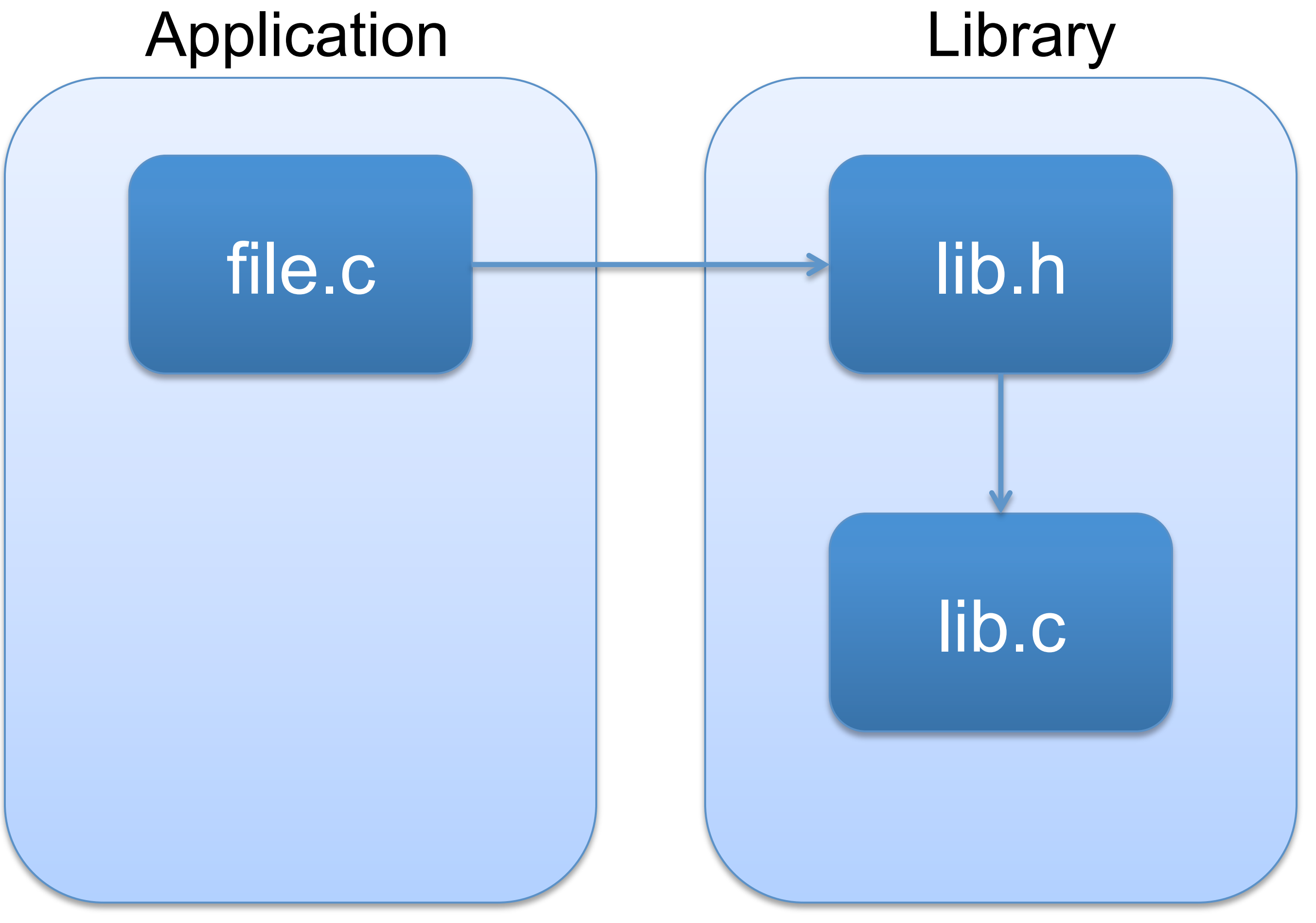


BMC for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)
- Add goals indicating the **coverage**
  - *location, branch, decision, condition and path*
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**
- Solve the formula iteratively to cover different goals
- Interpret the solution to figure out the **input conditions**
- Spit those input conditions out as a test case



Coverage Test Generation Example



Coverage Test Generation Example

file.c

```c
#include "lib.h"

int64_t nondet_int64_t();
int main() {
  int64_t a = nondet_int64_t();
  int64_t b = nondet_int64_t();
  int64_t r = nondet_int64_t();
  if (mul(a, b, &r)) {
    __ESBMC_assert(r == a * b, "Expected result from multiplication");
  }
  return 0;
}
```

Coverage Test Generation Example

lib.c

```
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    *res = 0;
    return 1;
  } else if(a == 1) {
    *res = b;
    return 1;
  } else if(b == 1) {
    *res = a;
    return 1;
  }
  *res = a * b; // there exists an overflow
  return 1;
}
```

Coverage Test Generation Example

lib.h

```
#include<stdint.h>
_Bool mul(const int64_t a, const int64_t b, int64_t *res);
```

```
esbmc main.c lib/lib.c --error-label GOALX -I lib/
```

Program Instrumentation

```
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    GOAL1:;
    *res = 0;
    return 1;
  } else if(a == 1) {
    GOAL2:;
    *res = b;
    return 1;
  } else if(b == 1) {
    GOAL3:;
    *res = a;
    return 1;
  }
GOAL4:;
  *res = a * b; // there exists an overflow
  return 1;
}
```

Program Instrumentation (Goal1)

```
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    GOAL1:;
    *res = 0;
    return 1;
  } else if(a == 1) {
    GOAL2:;
    *res = b;
    return 1;
  } else if(b == 1) {
    GOAL3:;
    *res = a;
    return 1;
  }
GOAL4:;
  *res = a * b; // there exists an overflow
  return 1;
}
```

Generate Test Case for Goal1

```
esbmc main.c lib/lib.c --error-label GOAL1 -I lib/
```

```
Counterexample:

State 1 file main.c line 5 function main thread 0
----------------------------------------------------
  a = 1 (00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000001)

State 2 file main.c line 6 function main thread 0
----------------------------------------------------
  b = 0 (00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000)

State 3 file lib.c line 5 function mul thread 0
----------------------------------------------------
Violated property:
  file lib.c line 5 function mul
  error label
  0
```

Program Instrumentation (Goal2)

```
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    GOAL1:;
    *res = 0;
    return 1;
  } else if(a == 1) {
    GOAL2:;
    *res = b;
    return 1;
  } else if(b == 1) {
    GOAL3:;
    *res = a;
    return 1;
  }
GOAL4:;
  *res = a * b; // there exists an overflow
  return 1;
}
```

Generate Test Case for Goal2

```
esbmc main.c lib/lib.c --error-label GOAL2 -I lib/
```

```
Counterexample:

State 1 file main.c line 5 function main thread 0
----------------------------------------------------
  a = 1 (00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000001)

State 2 file main.c line 6 function main thread 0
----------------------------------------------------
  b = 1 (00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000001)

State 3 file lib.c line 9 function mul thread 0
----------------------------------------------------
Violated property:
  file lib.c line 9 function mul
  error label
  0
```

Program Instrumentation (Goal3)

```c
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    GOAL1:;
    *res = 0;
    return 1;
  } else if(a == 1) {
    GOAL2:;
    *res = b;
    return 1;
  } else if(b == 1) {
    GOAL3:;
    *res = a;
    return 1;
  }
GOAL4:;
  *res = a * b; // there exists an overflow
  return 1;
}
```

Generate Test Case for Goal3

```
esbmc main.c lib/lib.c --error-label GOAL3 -I lib/
```

```
Counterexample:

State 1 file main.c line 5 function main thread 0
----------------------------------------------------
  a = -4537113969113143794 (11000001 00001000 11101110
11100010 00111101 10001100 01100110 00001110)

State 2 file main.c line 6 function main thread 0
----------------------------------------------------
  b = 1 (00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000001)

State 3 file lib.c line 13 function mul thread 0
----------------------------------------------------
Violated property:
  file lib.c line 13 function mul
  error label
  0
```

Program Instrumentation (Goal4)

```c
#include "lib.h"
_Bool mul(int64_t a, int64_t b, int64_t *res) {
  // Trivial cases
  if((a == 0) || (b == 0)) {
    GOAL1:;
    *res = 0;
    return 1;
  } else if(a == 1) {
    GOAL2:;
    *res = b;
    return 1;
  } else if(b == 1) {
    GOAL3:;
    *res = a;
    return 1;
  }
GOAL4:;
  *res = a * b; // there exists an overflow
  return 1;
}
```

Generate Test Case for Goal4

```
esbmc main.c lib/lib.c --error-label GOAL4 -I lib/
```

```
Counterexample:

State 1 file main.c line 5 function main thread 0
----------------------------------------------------
  a = 6917247552664371199 (01011111 11111110 11111111 11111111
11111111 11111111 11111111 11111111)

State 2 file main.c line 6 function main thread 0
----------------------------------------------------
  b = -1 (11111111 11111111 11111111 11111111 11111111
11111111 11111111 11111111)

State 3 file lib.c line 17 function mul thread 0
----------------------------------------------------
Violated property:
  file lib.c line 17 function mul
  error label
  0
```

Generate Test Case for Overflow

```
esbmc main.c lib/lib.c --overflow-check  -I lib/
```

```
Counterexample:

State 1 file main.c line 5 function main thread 0
----------------------------------------------------
  a = 4623516855184146434 (01000000 00101010 00001000
00010101 01010110 01001000 01000000 00000010)

State 2 file main.c line 6 function main thread 0
----------------------------------------------------
  b = 3 (00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000011)

State 3 file lib.c line 18 function mul thread 0
----------------------------------------------------
Violated property:
  file lib.c line 18 function mul
  arithmetic overflow on mul
  !overflow("*", a, b)
```

Summary

- Cost/precision tradeoffs
  - Blackbox is lightweight, easy and fast, but weak coverage
  - Whitebox is smarter but complex and slower
  - Recent "**semi-whitebox**" approaches
    - Less smart but more lightweight: **Flayer** (taint-flow analysis, may generate false alarms), **Bunny-the-fuzzer** (taint-flow, source-based, heuristics to fuzz based on input usage), **autodafe**, etc.

Summary

- Cost/precision tradeoffs
  - Blackbox is lightweight, easy and fast, but weak coverage
  - Whitebox is smarter but complex and slower
  - Recent "**semi-whitebox**" approaches
    - Less smart but more lightweight: **Flayer** (taint-flow analysis, may generate false alarms), **Bunny-the-fuzzer** (taint-flow, source-based, heuristics to fuzz based on input usage), **autodafe**, etc.
- Which is more effective at finding bugs? It depends…
  - Many apps are buggy; any form of fuzzing finds bugs!
  - Once low-hanging bugs are gone, fuzzing must become smarter: use whitebox and/or user-provided guidance (grammars, etc.)

Summary

- Cost/precision tradeoffs
  - Blackbox is lightweight, easy and fast, but weak coverage
  - Whitebox is smarter but complex and slower
  - Recent "**semi-whitebox**" approaches
    - Less smart but more lightweight: **Flayer** (taint-flow analysis, may generate false alarms), **Bunny-the-fuzzer** (taint-flow, source-based, heuristics to fuzz based on input usage), **autodafe**, etc.
- Which is more effective at finding bugs? It depends…
  - Many apps are buggy; any form of fuzzing finds bugs!
  - Once low-hanging bugs are gone, fuzzing must become smarter: use whitebox and/or user-provided guidance (grammars, etc.)
- Bottom line: in practice, **use both**!