DART: Directed Automated Random Testing

PLDI 2005

Patrice Godefroid[1] Nils Klarlund[1] Koushik Sen[2]

[1]Bell Laboratories, Lucent Technologies

[2]University of Illinois at Urbana-Champaign

November 10, 2015

Presented by Markus

Introduction

- Testing makes up 50% of software development cost

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

- Testing makes up 50% of software development cost
- Failures cost $60 billion per year in USA alone

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

- Testing makes up 50% of software development cost
- Failures cost $60 billion per year in USA alone
- Software testing is important

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

- Testing makes up 50% of software development cost
- Failures cost $60 billion per year in USA alone
- Software testing is important
- Software testing is. . .

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

- Testing makes up 50% of software development cost
- Failures cost $60 billion per year in USA alone
- Software testing is important
- Software testing is. . .
  - Hard

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

- Testing makes up 50% of software development cost
- Failures cost $60 billion per year in USA alone
- Software testing is important
- Software testing is...
  - Hard
  - Boring

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

- Testing makes up 50% of software development cost
- Failures cost $60 billion per year in USA alone
- Software testing is important
- Software testing is...
  - Hard
  - Boring
  - Tedious

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

- Testing makes up 50% of software development cost
- Failures cost $60 billion per year in USA alone
- Software testing is important
- Software testing is...
  - Hard
  - Boring
  - Tedious
- *Automated* techniques

http://vignette4.wikia.nocookie.net/spongebob/images/9/9f/Money_Krabs_CS.jpg

- Overall, testing makes up around fifty percent of the cost of developing software
- Complementing this is the fact that software failures in the USA cost around 60 billion dollars per year
- So, because people like money, software testing is important in order to both reduce the testing cost and prevent costly failures
- But, software testing, from a developers standpoint, is also hard, boring, and tedious
- Because of this, techniques to automatically test a program can help reduce developer burden and costs

Introduction

```
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Automated random testing

- To better illustrate why program testing is hard, and the difficulties with current automated techniques, we'll look at this example program
- Here, we have the function h which we would like to to test. We've encoded an error statement in h using the abort statement
- There are two conditions guarding the reachability of the abort statement: x must not be equal to y and the result of calling mul2 on x must be equal to x + 10
- Random testing is one automated testing technique: it simply applies random inputs to the function under test with hopes to execute different paths
- Random testing is good since it requires very low overhead but it often has difficulty exercising new paths within the program
- Specifically, if we examine look at a condition such as x equal to 10, with 32 bit integers there is a 2^{32} chance to guess this correctly
- Obviously, with such a low probability, random testing will likely end up having low coverage on this function
- An alternative approach is to use what the authors refer to as directed testing
- In this way, the inputs required to reach a specific point in the program are specified as a set of constraints who's satisfiability represent inputs to reach a certain location

Introduction

```c
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Automated random testing
  - Hard to guess constraints (`x == 10`)

- To better illustrate why program testing is hard, and the difficulties with current automated techniques, we'll look at this example program
- Here, we have the function h which we would like to to test. We've encoded an error statement in h using the abort statement
- There are two conditions guarding the reachability of the abort statement: x must not be equal to y and the result of calling mul2 on x must be equal to x + 10
- Random testing is one automated testing technique: it simply applies random inputs to the function under test with hopes to execute different paths
- Random testing is good since it requires very low overhead but it often has difficulty exercising new paths within the program
- Specifically, if we examine look at a condition such as x equal to 10, with 32 bit integers there is a 2^{32} chance to guess this correctly
- Obviously, with such a low probability, random testing will likely end up having low coverage on this function
- An alternative approach is to use what the authors refer to as directed testing
- In this way, the inputs required to reach a specific point in the program are specified as a set of constraints who's satisfiability represent inputs to reach a certain location

Introduction

```
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Automated random testing
  - Hard to guess constraints (x == 10)
- Directed random testing

- To better illustrate why program testing is hard, and the difficulties with current automated techniques, we'll look at this example program
- Here, we have the function h which we would like to to test. We've encoded an error statement in h using the abort statement
- There are two conditions guarding the reachability of the abort statement: x must not be equal to y and the result of calling mul2 on x must be equal to $x + 10$
- Random testing is one automated testing technique: it simply applies random inputs to the function under test with hopes to execute different paths
- Random testing is good since it requires very low overhead but it often has difficulty exercising new paths within the program
- Specifically, if we examine look at a condition such as x equal to 10, with 32 bit integers there is a 2^{32} chance to guess this correctly
- Obviously, with such a low probability, random testing will likely end up having low coverage on this function
- An alternative approach is to use what the authors refer to as directed testing
- In this way, the inputs required to reach a specific point in the program are specified as a set of constraints who's satisfiability represent inputs to reach a certain location

Introduction

```c
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Automated random testing
  - Hard to guess constraints (`x == 10`)
- Directed random testing
  - Specify reachability as *constraints*

- To better illustrate why program testing is hard, and the difficulties with current automated techniques, we'll look at this example program
- Here, we have the function h which we would like to to test. We've encoded an error statement in h using the abort statement
- There are two conditions guarding the reachability of the abort statement: x must not be equal to y and the result of calling mul2 on x must be equal to x + 10
- Random testing is one automated testing technique: it simply applies random inputs to the function under test with hopes to execute different paths
- Random testing is good since it requires very low overhead but it often has difficulty exercising new paths within the program
- Specifically, if we examine look at a condition such as x equal to 10, with 32 bit integers there is a 2^{32} chance to guess this correctly
- Obviously, with such a low probability, random testing will likely end up having low coverage on this function
- An alternative approach is to use what the authors refer to as directed testing
- In this way, the inputs required to reach a specific point in the program are specified as a set of constraints who's satisfiability represent inputs to reach a certain location

Introduction

```c
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Input One:
  $x = 20, y = 1000$

- To better understand this concept of directed testing, we'll continue looking at this example
- Consider we randomly generate the following inputs to h: x equal to 20 and y equal to 1000
- With this input, the first branch, x not equal to y, will be taken, but the second one will not since the result of mul2 returns 40 and 40 is not equal to 30
- Given this programs execution, we can capture its path constraint: the path constraint is a logical formula capturing all program inputs resulting in the same path
- Specifically, this path constraint specifies that x is not equal to y and 2x is not equal to x + 10: intuitively, we can see these conditions represent the first branch being taken and the second one not being taken
- Since our goal is to increase testing coverage of the function, we'd like to direct the tester to explore a new path through the function
- To do this, we can negate the last condition in the previous constraint, in other words, try to find an input to satisfy the first and second branch conditions
- Passing this equation to a solver, we can get a solution that x equals 10 and y equals 1000 which are valid inputs to reach the abort statement and find the bug

Introduction

```
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Input One: $x = 20, y = 1000$
  - Second branch not taken: $40 \neq 20 + 10$

- To better understand this concept of directed testing, we'll continue looking at this example
- Consider we randomly generate the following inputs to h: x equal to 20 and y equal to 1000
- With this input, the first branch, x not equal to y, will be taken, but the second one will not since the result of mul2 returns 40 and 40 is not equal to 30
- Given this programs execution, we can capture its path constraint: the path constraint is a logical formula capturing all program inputs resulting in the same path
- Specifically, this path constraint specifies that x is not equal to y and 2x is not equal to x + 10: intuitively, we can see these conditions represent the first branch being taken and the second one not being taken
- Since our goal is to increase testing coverage of the function, we'd like to direct the tester to explore a new path through the function
- To do this, we can negate the last condition in the previous constraint, in other words, try to find an input to satisfy the first and second branch conditions
- Passing this equation to a solver, we can get a solution that x equals 10 and y equals 1000 which are valid inputs to reach the abort statement and find the bug

Introduction

```
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Input One: $x = 20, y = 1000$
  - Second branch not taken: $40 \neq 20 + 10$
- Path constraint: $(x \neq y) \wedge (2x \neq x + 10)$

- To better understand this concept of directed testing, we'll continue looking at this example
- Consider we randomly generate the following inputs to h: x equal to 20 and y equal to 1000
- With this input, the first branch, x not equal to y, will be taken, but the second one will not since the result of mul2 returns 40 and 40 is not equal to 30
- Given this programs execution, we can capture its path constraint: the path constraint is a logical formula capturing all program inputs resulting in the same path
- Specifically, this path constraint specifies that x is not equal to y and 2x is not equal to x + 10: intuitively, we can see these conditions represent the first branch being taken and the second one not being taken
- Since our goal is to increase testing coverage of the function, we'd like to direct the tester to explore a new path through the function
- To do this, we can negate the last condition in the previous constraint, in other words, try to find an input to satisfy the first and second branch conditions
- Passing this equation to a solver, we can get a solution that x equals 10 and y equals 1000 which are valid inputs to reach the abort statement and find the bug

Introduction

```
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
  }
}
```

- Input One: $x = 20, y = 1000$
  - Second branch not taken: $40 \neq 20 + 10$
- Path constraint: $(x \neq y) \wedge (2x \neq x + 10)$
- *Direct* tester to new paths

- To better understand this concept of directed testing, we'll continue looking at this example
- Consider we randomly generate the following inputs to h: x equal to 20 and y equal to 1000
- With this input, the first branch, x not equal to y, will be taken, but the second one will not since the result of mul2 returns 40 and 40 is not equal to 30
- Given this programs execution, we can capture its path constraint: the path constraint is a logical formula capturing all program inputs resulting in the same path
- Specifically, this path constraint specifies that x is not equal to y and 2x is not equal to x + 10: intuitively, we can see these conditions represent the first branch being taken and the second one not being taken
- Since our goal is to increase testing coverage of the function, we'd like to direct the tester to explore a new path through the function
- To do this, we can negate the last condition in the previous constraint, in other words, try to find an input to satisfy the first and second branch conditions
- Passing this equation to a solver, we can get a solution that x equals 10 and y equals 1000 which are valid inputs to reach the abort statement and find the bug

Introduction

```
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
}
```

- Input One:
  $x = 20, y = 1000$
  - Second branch not taken: $40 \neq 20 + 10$
- Path constraint:
  $(x \neq y) \wedge (2x \neq x + 10)$
- *Direct* tester to new paths
  - Alter path constraint & solve

- To better understand this concept of directed testing, we'll continue looking at this example
- Consider we randomly generate the following inputs to h: x equal to 20 and y equal to 1000
- With this input, the first branch, x not equal to y, will be taken, but the second one will not since the result of mul2 returns 40 and 40 is not equal to 30
- Given this programs execution, we can capture its path constraint: the path constraint is a logical formula capturing all program inputs resulting in the same path
- Specifically, this path constraint specifies that x is not equal to y and 2x is not equal to x + 10: intuitively, we can see these conditions represent the first branch being taken and the second one not being taken
- Since our goal is to increase testing coverage of the function, we'd like to direct the tester to explore a new path through the function
- To do this, we can negate the last condition in the previous constraint, in other words, try to find an input to satisfy the first and second branch conditions
- Passing this equation to a solver, we can get a solution that x equals 10 and y equals 1000 which are valid inputs to reach the abort statement and find the bug

Introduction

```c
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
  }
}
```

- Input One: $x = 20, y = 1000$
  - Second branch not taken: $40 \neq 20 + 10$
- Path constraint: $(x \neq y) \wedge (2x \neq x + 10)$
- *Direct* tester to new paths
  - Alter path constraint & solve
- New constraint: $(x \neq y) \wedge (2x = x + 10)$

- To better understand this concept of directed testing, we'll continue looking at this example
- Consider we randomly generate the following inputs to h: x equal to 20 and y equal to 1000
- With this input, the first branch, x not equal to y, will be taken, but the second one will not since the result of mul2 returns 40 and 40 is not equal to 30
- Given this programs execution, we can capture its path constraint: the path constraint is a logical formula capturing all program inputs resulting in the same path
- Specifically, this path constraint specifies that x is not equal to y and 2x is not equal to x + 10: intuitively, we can see these conditions represent the first branch being taken and the second one not being taken
- Since our goal is to increase testing coverage of the function, we'd like to direct the tester to explore a new path through the function
- To do this, we can negate the last condition in the previous constraint, in other words, try to find an input to satisfy the first and second branch conditions
- Passing this equation to a solver, we can get a solution that x equals 10 and y equals 1000 which are valid inputs to reach the abort statement and find the bug

Introduction

```c
int mul2(int x) {
  return 2 * x;
}
int h(int x, int y) {
  if (x != y) {
    if (mul2(x) == x + 10) {
      abort();
    }
  }
}
```

- Input One: $x = 20, y = 1000$
  - Second branch not taken: $40 \neq 20 + 10$
- Path constraint: $(x \neq y) \land (2x \neq x + 10)$
- *Direct* tester to new paths
  - Alter path constraint & solve
- New constraint: $(x \neq y) \land (2x = x + 10)$
  - $x = 10 \land y = 1000$

- To better understand this concept of directed testing, we'll continue looking at this example
- Consider we randomly generate the following inputs to h: x equal to 20 and y equal to 1000
- With this input, the first branch, x not equal to y, will be taken, but the second one will not since the result of mul2 returns 40 and 40 is not equal to 30
- Given this programs execution, we can capture its path constraint: the path constraint is a logical formula capturing all program inputs resulting in the same path
- Specifically, this path constraint specifies that x is not equal to y and 2x is not equal to x + 10: intuitively, we can see these conditions represent the first branch being taken and the second one not being taken
- Since our goal is to increase testing coverage of the function, we'd like to direct the tester to explore a new path through the function
- To do this, we can negate the last condition in the previous constraint, in other words, try to find an input to satisfy the first and second branch conditions
- Passing this equation to a solver, we can get a solution that x equals 10 and y equals 1000 which are valid inputs to reach the abort statement and find the bug

Contributions

- Random testing + directed testing

- This brings us to the authors contributions
- The authors present a framework combining random testing with directed testing
- The approach works just as in the previous example: they first randomly apply function inputs, gather a set of path constraints on an explored trace, and then use a solver to generate new inputs guiding the program along a new path
- Along with this testing technique, they also present a technique to identify interfaces, or, locations which should be tested, in the program
- In this way, the authors analysis becomes fully automated without requiring the developers to do anything

Contributions

- Random testing + directed testing
- Randomly apply function inputs

- This brings us to the authors contributions
- The authors present a framework combining random testing with directed testing
- The approach works just as in the previous example: they first randomly apply function inputs, gather a set of path constraints on an explored trace, and then use a solver to generate new inputs guiding the program along a new path
- Along with this testing technique, they also present a technique to identify interfaces, or, locations which should be tested, in the program
- In this way, the authors analysis becomes fully automated without requiring the developers to do anything

Contributions

- Random testing + directed testing
- Randomly apply function inputs
- Gather path constraints on a trace

- This brings us to the authors contributions
- The authors present a framework combining random testing with directed testing
- The approach works just as in the previous example: they first randomly apply function inputs, gather a set of path constraints on an explored trace, and then use a solver to generate new inputs guiding the program along a new path
- Along with this testing technique, they also present a technique to identify interfaces, or, locations which should be tested, in the program
- In this way, the authors analysis becomes fully automated without requiring the developers to do anything

Contributions

- Random testing + directed testing
- Randomly apply function inputs
- Gather path constraints on a trace
- Use solver to find new inputs

- This brings us to the authors contributions
- The authors present a framework combining random testing with directed testing
- The approach works just as in the previous example: they first randomly apply function inputs, gather a set of path constraints on an explored trace, and then use a solver to generate new inputs guiding the program along a new path
- Along with this testing technique, they also present a technique to identify interfaces, or, locations which should be tested, in the program
- In this way, the authors analysis becomes fully automated without requiring the developers to do anything

Contributions

- Random testing + directed testing
- Randomly apply function inputs
- Gather path constraints on a trace
- Use solver to find new inputs
- Static method to identify interfaces

- This brings us to the authors contributions
- The authors present a framework combining random testing with directed testing
- The approach works just as in the previous example: they first randomly apply function inputs, gather a set of path constraints on an explored trace, and then use a solver to generate new inputs guiding the program along a new path
- Along with this testing technique, they also present a technique to identify interfaces, or, locations which should be tested, in the program
- In this way, the authors analysis becomes fully automated without requiring the developers to do anything

Contributions

- Random testing + directed testing
- Randomly apply function inputs
- Gather path constraints on a trace
- Use solver to find new inputs
- Static method to identify interfaces
- Fully automated

- This brings us to the authors contributions
- The authors present a framework combining random testing with directed testing
- The approach works just as in the previous example: they first randomly apply function inputs, gather a set of path constraints on an explored trace, and then use a solver to generate new inputs guiding the program along a new path
- Along with this testing technique, they also present a technique to identify interfaces, or, locations which should be tested, in the program
- In this way, the authors analysis becomes fully automated without requiring the developers to do anything

Overview

Next, I'll go over how the authors generate path constraints during testing

Path Constraints: Overview

1. Execute the program with random inputs

- Here again is an overivew of the exploration technique used by the authors
- First, they execute the program with random inputs
- During the execution of the program, they collect the path constraints visited by the dynamic execution
- To collect these paths constraints, they instrument each statement in the program and model the semantics of the statements
- Next, given the path constraints from one execution, they negate one of the branches in the path constraint and pass the formula to a solver
- The solver then attempts to find a valuation of the program inputs such that the path constraint is satisfied, or, in other words, values of the program inputs such that the new path is expored
- They then use these newly generated inputs to the program and re-execute the program and repeat the process
- To make this more clear, I'll go over an example

Path Constraints: Overview

1. Execute the program with random inputs
2. Collect path-constraints of execution

- Here again is an overivew of the exploration technique used by the authors
- First, they execute the program with random inputs
- During the execution of the program, they collect the path constraints visited by the dynamic execution
- To collect these paths constraints, they instrument each statement in the program and model the semantics of the statements
- Next, given the path constraints from one execution, they negate one of the branches in the path constraint and pass the formula to a solver
- The solver then attempts to find a valuation of the program inputs such that the path constraint is satisfied, or, in other words, values of the program inputs such that the new path is expored
- They then use these newly generated inputs to the program and re-execute the program and repeat the process
- To make this more clear, I'll go over an example

Path Constraints: Overview

1. Execute the program with random inputs
2. Collect path-constraints of execution
3. Negate a condition to generate new inputs

- Here again is an overivew of the exploration technique used by the authors
- First, they execute the program with random inputs
- During the execution of the program, they collect the path constraints visited by the dynamic execution
- To collect these paths constraints, they instrument each statement in the program and model the semantics of the statements
- Next, given the path constraints from one execution, they negate one of the branches in the path constraint and pass the formula to a solver
- The solver then attempts to find a valuation of the program inputs such that the path constraint is satisfied, or, in other words, values of the program inputs such that the new path is expored
- They then use these newly generated inputs to the program and re-execute the program and repeat the process
- To make this more clear, I'll go over an example

Path Constraints: Overview

1. Execute the program with random inputs
2. Collect path-constraints of execution
3. Negate a condition to generate new inputs
4. Repeat

- Here again is an overivew of the exploration technique used by the authors
- First, they execute the program with random inputs
- During the execution of the program, they collect the path constraints visited by the dynamic execution
- To collect these paths constraints, they instrument each statement in the program and model the semantics of the statements
- Next, given the path constraints from one execution, they negate one of the branches in the path constraint and pass the formula to a solver
- The solver then attempts to find a valuation of the program inputs such that the path constraint is satisfied, or, in other words, values of the program inputs such that the new path is expored
- They then use these newly generated inputs to the program and re-execute the program and repeat the process
- To make this more clear, I'll go over an example

Path Constraints: Example (1)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```


- Here is an example program which is slightly more complicated than the one we previously looked at because it has some side effects.
- To understand the path-constraint generation approach, we'll go through this program line-by-line and look at how it evolves symbolically
- First, if we look at the concrete execution with these inputs, the first branch is not taken since x is not equal to z. So, the first test halts after the check of the first branch
- During the concrete execution of the program, the authors build a symbolic representation of all the variables
- Before the execution of the function, the program inputs are unconstrained, here, I assume 32 bit integers
- After executing line 2, the value of z is updated to be the value of y
- Similarly, the value of c1 is updated to be the value of the expression x equal to y. Notice that this sets the value of c to be the boolean value represented by the expression z equal to y
- Finally, since during the concrete execution the branch was not taken we negate the condition in the branch to generate the path constraint for the first run

Path Constraints: Example (1)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input: $x = 10, y = 20$

- Here is an example program which is slightly more complicated than the one we previously looked at because it has some side effects.
- To understand the path-constraint generation approach, we'll go through this program line-by-line and look at how it evolves symbolically
- First, if we look at the concrete execution with these inputs, the first branch is not taken since x is not equal to z. So, the first test halts after the check of the first branch
- During the concrete execution of the program, the authors build a symbolic representation of all the variables
- Before the execution of the function, the program inputs are unconstrained, here, I assume 32 bit integers
- After executing line 2, the value of z is updated to be the value of y
- Similarly, the value of c1 is updated to be the value of the expression x equal to y. Notice that this sets the value of c to be the boolean value represented by the expression z equal to y
- Finally, since during the concrete execution the branch was not taken we negate the condition in the branch to generate the path constraint for the first run

Path Constraints: Example (1)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
  - $z = 20 \rightarrow x \neq z$

- Here is an example program which is slightly more complicated than the one we previously looked at because it has some side effects.
- To understand the path-constraint generation approach, we'll go through this program line-by-line and look at how it evolves symbolically
- First, if we look at the concrete execution with these inputs, the first branch is not taken since x is not equal to z. So, the first test halts after the check of the first branch
- During the concrete execution of the program, the authors build a symbolic representation of all the variables
- Before the execution of the function, the program inputs are unconstrained, here, I assume 32 bit integers
- After executing line 2, the value of z is updated to be the value of y
- Similarly, the value of c1 is updated to be the value of the expression x equal to y. Notice that this sets the value of c to be the boolean value represented by the expression z equal to y
- Finally, since during the concrete execution the branch was not taken we negate the condition in the branch to generate the path constraint for the first run

Path Constraints: Example (1)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input: $x = 10, y = 20$
- Initially:

$$-2^{31} \leq x \leq 2^{31} - 1$$
$$\land -2^{31} \leq y \leq 2^{31} - 1$$

- Here is an example program which is slightly more complicated than the one we previously looked at because it has some side effects.
- To understand the path-constraint generation approach, we'll go through this program line-by-line and look at how it evolves symbolically
- First, if we look at the concrete execution with these inputs, the first branch is not taken since x is not equal to z. So, the first test halts after the check of the first branch
- During the concrete execution of the program, the authors build a symbolic representation of all the variables
- Before the execution of the function, the program inputs are unconstrained, here, I assume 32 bit integers
- After executing line 2, the value of z is updated to be the value of y
- Similarly, the value of c1 is updated to be the value of the expression x equal to y. Notice that this sets the value of c to be the boolean value represented by the expression z equal to y
- Finally, since during the concrete execution the branch was not taken we negate the condition in the branch to generate the path constraint for the first run

Path Constraints: Example (1)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
- After line 2:

$$
\begin{aligned}
&-2^{31} \leq x \leq 2^{31} - 1 \\
&\wedge -2^{31} \leq y \leq 2^{31} - 1 \\
&\wedge z := y
\end{aligned}
$$

- Here is an example program which is slightly more complicated than the one we previously looked at because it has some side effects.
- To understand the path-constraint generation approach, we'll go through this program line-by-line and look at how it evolves symbolically
- First, if we look at the concrete execution with these inputs, the first branch is not taken since x is not equal to z. So, the first test halts after the check of the first branch
- During the concrete execution of the program, the authors build a symbolic representation of all the variables
- Before the execution of the function, the program inputs are unconstrained, here, I assume 32 bit integers
- After executing line 2, the value of z is updated to be the value of y
- Similarly, the value of c1 is updated to be the value of the expression x equal to y. Notice that this sets the value of c to be the boolean value represented by the expression z equal to y
- Finally, since during the concrete execution the branch was not taken we negate the condition in the branch to generate the path constraint for the first run

Path Constraints: Example (1)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
- After line 3:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z)
\end{aligned}
$$

- Here is an example program which is slightly more complicated than the one we previously looked at because it has some side effects.
- To understand the path-constraint generation approach, we'll go through this program line-by-line and look at how it evolves symbolically
- First, if we look at the concrete execution with these inputs, the first branch is not taken since x is not equal to z. So, the first test halts after the check of the first branch
- During the concrete execution of the program, the authors build a symbolic representation of all the variables
- Before the execution of the function, the program inputs are unconstrained, here, I assume 32 bit integers
- After executing line 2, the value of z is updated to be the value of y
- Similarly, the value of c1 is updated to be the value of the expression x equal to y. Notice that this sets the value of c to be the boolean value represented by the expression z equal to y
- Finally, since during the concrete execution the branch was not taken we negate the condition in the branch to generate the path constraint for the first run

Path Constraints: Example (1)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input: $x = 10, y = 20$
- After line 3:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z)
\end{aligned}
$$

- Path constraint: $\neg c_1$

- Here is an example program which is slightly more complicated than the one we previously looked at because it has some side effects.
- To understand the path-constraint generation approach, we'll go through this program line-by-line and look at how it evolves symbolically
- First, if we look at the concrete execution with these inputs, the first branch is not taken since x is not equal to z. So, the first test halts after the check of the first branch
- During the concrete execution of the program, the authors build a symbolic representation of all the variables
- Before the execution of the function, the program inputs are unconstrained, here, I assume 32 bit integers
- After executing line 2, the value of z is updated to be the value of y
- Similarly, the value of c1 is updated to be the value of the expression x equal to y. Notice that this sets the value of c to be the boolean value represented by the expression z equal to y
- Finally, since during the concrete execution the branch was not taken we negate the condition in the branch to generate the path constraint for the first run

Path Constraints: Example (2)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 3:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z)
\end{aligned}
$$

- Old constraint: $\neg c_1$

- After generating the symbolic expression for the variables along with the path constraint, the next step is to generate a new input to the program in order to explore a new path
- Since we've only seen one branch, the only new choice we can make is to explore inside this branch, or, to find program inputs such that c_1 is true
- To do this, we use the symbolic values for all the variables and conjunct it with the path constraint we want to build a new logic formula
- Next, we can ask a solver to find a satisfying assignment to this formula: the satisfying assignment is a valuation of x and y such that all the constraints hold
- One such solution is that x and y are both equal to zero
- The key thing to notice is that the logic formula we've constructed is such that a satisfying assignment represents values of the inputs which are guaranteed to reach the branch we are interested in

Path Constraints: Example (2)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 3:

$$-2^{31} \leq x \leq 2^{31} - 1$$
$$\wedge -2^{31} \leq y \leq 2^{31} - 1$$
$$\wedge z := y$$
$$\wedge c_1 := (x = z)$$

- Old constraint: $\neg c_1$
- New constraint: c_1

- After generating the symbolic expression for the variables along with the path constraint, the next step is to generate a new input to the program in order to explore a new path
- Since we've only seen one branch, the only new choice we can make is to explore inside this branch, or, to find program inputs such that c_1 is true
- To do this, we use the symbolic values for all the variables and conjunct it with the path constraint we want to build a new logic formula
- Next, we can ask a solver to find a satisfying assignment to this formula: the satisfying assignment is a valuation of x and y such that all the constraints hold
- One such solution is that x and y are both equal to zero
- The key thing to notice is that the logic formula we've constructed is such that a satisfying assignment represents values of the inputs which are guaranteed to reach the branch we are interested in

Path Constraints: Example (2)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- Logic formula:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z) \\
& \wedge c_1
\end{aligned}
$$

- After generating the symbolic expression for the variables along with the path constraint, the next step is to generate a new input to the program in order to explore a new path
- Since we've only seen one branch, the only new choice we can make is to explore inside this branch, or, to find program inputs such that c_1 is true
- To do this, we use the symbolic values for all the variables and conjunct it with the path constraint we want to build a new logic formula
- Next, we can ask a solver to find a satisfying assignment to this formula: the satisfying assignment is a valuation of x and y such that all the constraints hold
- One such solution is that x and y are both equal to zero
- The key thing to notice is that the logic formula we've constructed is such that a satisfying assignment represents values of the inputs which are guaranteed to reach the branch we are interested in

Path Constraints: Example (2)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- Logic formula:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z) \\
& \wedge c_1
\end{aligned}
$$

- Satisfying assignment: $x = 0 \wedge y = 0$

- After generating the symbolic expression for the variables along with the path constraint, the next step is to generate a new input to the program in order to explore a new path
- Since we've only seen one branch, the only new choice we can make is to explore inside this branch, or, to find program inputs such that c_1 is true
- To do this, we use the symbolic values for all the variables and conjunct it with the path constraint we want to build a new logic formula
- Next, we can ask a solver to find a satisfying assignment to this formula: the satisfying assignment is a valuation of x and y such that all the constraints hold
- One such solution is that x and y are both equal to zero
- The key thing to notice is that the logic formula we've constructed is such that a satisfying assignment represents values of the inputs which are guaranteed to reach the branch we are interested in

Path Constraints: Example (3)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- Concrete input:
  $x = 0, y = 0$

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Path Constraints: Example (3)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- Concrete input: $x = 0, y = 0$
- `c1 = x == z = 1`

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Path Constraints: Example (3)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- Concrete input: $x = 0, y = 0$
- `c1 = x == z = 1`
- `t2 = x + 10 = 10`

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Path Constraints: Example (3)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 6:

$$
\begin{aligned}
&2^{31} \leq x \leq 2^{31} - 1 \\
&\quad \wedge\, 2^{31} \leq y \leq 2^{31} - 1 \\
&\quad \wedge\, z := y \\
&\quad \wedge\, c_1 := (x = z) \\
&\quad \wedge\, t_2 := x + 10 \\
&\quad \wedge\, c_2 := y = t_2
\end{aligned}
$$

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Path Constraints: Example (3)

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 6:

$$
\begin{aligned}
&2^{31} \leq x \leq 2^{31} - 1\\
&\quad \wedge 2^{31} \leq y \leq 2^{31} - 1\\
&\quad \wedge z := y\\
&\quad \wedge c_1 := (x = z)\\
&\quad \wedge t_2 := x + 10\\
&\quad \wedge c_2 := y = t_2
\end{aligned}
$$

- Path constraint: $c_1 \wedge \neg c_2$

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Path Constraints: Example (3)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- New constraint: $c_1 \wedge c_2$

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Path Constraints: Example (3)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- New constraint: $c_1 \land c_2$
- Logic formula:

$$
\begin{aligned}
& 2^{31} \leq x \leq 2^{31} - 1 \\
& \land 2^{31} \leq y \leq 2^{31} - 1 \\
& \land z := y \\
& \land c_1 := (x = z) \\
& \land t_2 := x + 10 \\
& \land c_2 := y = t_2 \\
& \land c_1 \land c_2
\end{aligned}
$$

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Path Constraints: Example (3)

```
int f(int x, int y) {
  int z = y;
  bool c1 = x == z;
  if (c1) {
    int t2 = x + 10;
    bool c2 = y == t2;
    if (c2) {
      abort();
    }
  }
}
```

- New constraint: $c_1 \land c_2$
- Logic formula:

$$
\begin{aligned}
&2^{31} \leq x \leq 2^{31} - 1 \\
&\land 2^{31} \leq y \leq 2^{31} - 1 \\
&\land z := y \\
&\land c_1 := (x = z) \\
&\land t_2 := x + 10 \\
&\land c_2 := y = t_2 \\
&\land c_1 \land c_2
\end{aligned}
$$

- Unsatisfiable! (The error is unreachable)

- On the next iteration, we use the inputs we obtained previously to re-execute the program concretely
- During the concrete execution, we enter the first if-branch, then, we calculate the value of t2 which is x plus ten which evaluates to 10
- The value of c2 check is y is equal to t2 which evaluates to false
- So, the results of the second iteration are that the first branch is taken and the second branch is not taken
- Again, during the concrete execution we can generate a symbolic representation of the program. The symbolic representation this time is the same as in the previous iteration except it includes the evaluations of t2 and c2
- Again, this execution has a path constraint which is c1 and not c2. To generate the next path constraint we again flip one of the conditions and produce a new logic formula with the desired path conditions we want
- As a human, solving the constraints on the input variables to reach this location is already, at least for me, becoming non-trivial
- Luckily, we can use a solver to solve this formula: the result from the solver is that the formula is unsatisfiable: this means that there does not exist a value for the inputs to cause the abort to be reached
- For this function at least, the procedure is sound: we've formally proved that the abort statement in this function can never be reached

Implementation Intuition

- Transfer functions

- Now that I've gone over an example of their technique, I'll go over a high level intuition of how their technique works and try to relate it back to stuff we've seen so far
- Like most of the analyses we've seen so far, their technique uses transfer functions
- To keep track of the symbolic values of all the variables, the authors define transfer functions for all statements in the program
- For example, if we encounter an assignment statement during the execution, we use a transfer function which takes as input a symbolic representation, S, and returns a new symbolic representation which is the same as S except the value of x is assigned to z
- Defining transfer functions for every type of statement in the program allows for the analysis to operate on arbitrary sequences of expressions

Implementation Intuition

- Transfer functions
  - Function from symbolic equation to symbolic equation

- Now that I've gone over an example of their technique, I'll go over a high level intuition of how their technique works and try to relate it back to stuff we've seen so far
- Like most of the analyses we've seen so far, their technique uses transfer functions
- To keep track of the symbolic values of all the variables, the authors define transfer functions for all statements in the program
- For example, if we encounter an assignment statement during the execution, we use a transfer function which takes as input a symbolic representation, S, and returns a new symbolic representation which is the same as S except the value of x is assigned to z
- Defining transfer functions for every type of statement in the program allows for the analysis to operate on arbitrary sequences of expressions

Implementation Intuition

- Transfer functions
  - Function from symbolic equation to symbolic equation
  - $\mathcal{S} \rightarrow \mathcal{S}$

- Now that I've gone over an example of their technique, I'll go over a high level intuition of how their technique works and try to relate it back to stuff we've seen so far
- Like most of the analyses we've seen so far, their technique uses transfer functions
- To keep track of the symbolic values of all the variables, the authors define transfer functions for all statements in the program
- For example, if we encounter an assignment statement during the execution, we use a transfer function which takes as input a symbolic representation, S, and returns a new symbolic representation which is the same as S except the value of x is assigned to z
- Defining transfer functions for every type of statement in the program allows for the analysis to operate on arbitrary sequences of expressions

Implementation Intuition

- Transfer functions
  - Function from symbolic equation to symbolic equation
  - $\mathcal{S} \rightarrow \mathcal{S}$
- Evaluate: `z = x`

- Now that I've gone over an example of their technique, I'll go over a high level intuition of how their technique works and try to relate it back to stuff we've seen so far
- Like most of the analyses we've seen so far, their technique uses transfer functions
- To keep track of the symbolic values of all the variables, the authors define transfer functions for all statements in the program
- For example, if we encounter an assignment statement during the execution, we use a transfer function which takes as input a symbolic representation, S, and returns a new symbolic representation which is the same as S except the value of x is assigned to z
- Defining transfer functions for every type of statement in the program allows for the analysis to operate on arbitrary sequences of expressions

Implementation Intuition

- Transfer functions
  - Function from symbolic equation to symbolic equation
  - $\mathcal{S} \to \mathcal{S}$
- Evaluate: `z = x`
  - $\lambda S.S[\![z := x]\!]$

- Now that I've gone over an example of their technique, I'll go over a high level intuition of how their technique works and try to relate it back to stuff we've seen so far
- Like most of the analyses we've seen so far, their technique uses transfer functions
- To keep track of the symbolic values of all the variables, the authors define transfer functions for all statements in the program
- For example, if we encounter an assignment statement during the execution, we use a transfer function which takes as input a symbolic representation, S, and returns a new symbolic representation which is the same as S except the value of x is assigned to z
- Defining transfer functions for every type of statement in the program allows for the analysis to operate on arbitrary sequences of expressions

Soundness

- Programs may be infinite

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Soundness

- Programs may be infinite
  - Cannot have an infinitly long formulas

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Soundness

- Programs may be infinite
  - Cannot have an infinitly long formulas
- Solution: bound the depth of the search

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Soundness

- Programs may be infinite
  - Cannot have an infinitly long formulas
- Solution: bound the depth of the search
- Under-approximated analysis

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Soundness

- Programs may be infinite
  - Cannot have an infinitly long formulas
- Solution: bound the depth of the search
- Under-approximated analysis
  - Bug hunting

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Soundness

- Programs may be infinite
  - Cannot have an infinitly long formulas
- Solution: bound the depth of the search
- Under-approximated analysis
  - Bug hunting
  - Not proof generation

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Soundness

- Programs may be infinite
  - Cannot have an infinitly long formulas
- Solution: bound the depth of the search
- Under-approximated analysis
  - Bug hunting
  - Not proof generation
- No false alarms:

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Soundness

- Programs may be infinite
  - Cannot have an infinitly long formulas
- Solution: bound the depth of the search
- Under-approximated analysis
  - Bug hunting
  - Not proof generation
- No false alarms:
  - Detected bugs are guarnateed to exist in the actual program

- Since in general programs may be infinite, for example, in the presence of infinite loops, the analysis cannot generally handle all types of programs
- This is because we eventually need to produce a logic formula representing a path through the program: this logic formula cannot be infinitely long
- The solution to this problem is to only search through a bounded depth of a program
- As a result, the authors analysis, in general, is under-approximated
- This means it should be used for bug hunting and not proof generation
- However, because it is under-approximated, we have a nice side effect that the analysis has no false alarms
- This means that any bug which is detected by the algorithm is guaranteed to be a real bug

Overview

Now that I've gone over a high-level intution behind their approach, I'll present the experimental results

Test Bench

- Pentium III 800 MHz Processor

- The authors implemented their tool to test C programs
- They ran tests on a Pentium III processor running at 800 MHz
- They used a solver called lp solve to solve the constraint formulas
- And, they tested on three different programs: a small air conditioner controller example, a crypto protocol, and an open source library called oSIP

Test Bench

- Pentium III 800 MHz Processor
- lp_solve solver

- The authors implemented their tool to test C programs
- They ran tests on a Pentium III processor running at 800 MHz
- They used a solver called lp solve to solve the constraint formulas
- And, they tested on three different programs: a small air conditioner controller example, a crypto protocol, and an open source library called oSIP

Test Bench

- Pentium III 800 MHz Processor
- lp_solve solver
- CIL parser

- The authors implemented their tool to test C programs
- They ran tests on a Pentium III processor running at 800 MHz
- They used a solver called lp solve to solve the constraint formulas
- And, they tested on three different programs: a small air conditioner controller example, a crypto protocol, and an open source library called oSIP

Test Bench

- Pentium III 800 MHz Processor
- lp_solve solver
- CIL parser
- Three programs:

- The authors implemented their tool to test C programs
- They ran tests on a Pentium III processor running at 800 MHz
- They used a solver called lp solve to solve the constraint formulas
- And, they tested on three different programs: a small air conditioner controller example, a crypto protocol, and an open source library called oSIP

Test Bench

- Pentium III 800 MHz Processor
- lp_solve solver
- CIL parser
- Three programs:
  1. Air-Conditioner Controller

- The authors implemented their tool to test C programs
- They ran tests on a Pentium III processor running at 800 MHz
- They used a solver called lp solve to solve the constraint formulas
- And, they tested on three different programs: a small air conditioner controller example, a crypto protocol, and an open source library called oSIP

Test Bench

- Pentium III 800 MHz Processor
- lp_solve solver
- CIL parser
- Three programs:
  1. Air-Conditioner Controller
  2. Needham-Schroeder Protocol

- The authors implemented their tool to test C programs
- They ran tests on a Pentium III processor running at 800 MHz
- They used a solver called lp solve to solve the constraint formulas
- And, they tested on three different programs: a small air conditioner controller example, a crypto protocol, and an open source library called oSIP

Test Bench

- Pentium III 800 MHz Processor
- lp_solve solver
- CIL parser
- Three programs:
  1. Air-Conditioner Controller
  2. Needham-Schroeder Protocol
  3. oSIP Telephony Library

- The authors implemented their tool to test C programs
- They ran tests on a Pentium III processor running at 800 MHz
- They used a solver called lp solve to solve the constraint formulas
- And, they tested on three different programs: a small air conditioner controller example, a crypto protocol, and an open source library called oSIP

AC-Controller

```c
int is_room_hot, ac, is_door_closed;
void ac_controller(int message) {
  if (message == 0) is_room_hot = 1;
  if (message == 1) is_room_hot = 0;
  if (message == 2) {
    is_door_closed = 0;
    ac = 0;
  }
  if (message == 3) {
    is_door_closed = 1;
    if (is_room_hot) ac = 1;
  }
  if (is_room_hot && is_door_closed
      && !ac) {
    abort();
  }
}
```

- Random testing does not work

- First, we can look at the source code of the AC controller
- The source code is very small but makes a serves as a good comparison to randomized testing
- The program is buggy: the abort statement in the program is reachable under certain program inputs
- First, to understand how this function was run you need to imagine that this function can be called an arbitrary number of times with different values for message
- It is essentially representing a state machine which causes transitions based on the input to the function
- The abort statement in the program can be reached after applying two messages: first passing 3 and then passing 0
- Because this bug takes at least two messages to manifest, the chance for a random tester to find it is one out of 2 to the sixty four, which is obviously very close to zero
- DART on the other hand, finds the bug in less than one second

AC-Controller

```c
int is_room_hot, ac, is_door_closed;
void ac_controller(int message) {
  if (message == 0) is_room_hot = 1;
  if (message == 1) is_room_hot = 0;
  if (message == 2) {
    is_door_closed = 0;
    ac = 0;
  }
  if (message == 3) {
    is_door_closed = 1;
    if (is_room_hot) ac = 1;
  }
  if (is_room_hot && is_door_closed
      && !ac) {
    abort();
  }
}
```

- Random testing does not work
- $2^{32} \times 2^{32} = 2^{64}$ number of possibilities

- First, we can look at the source code of the AC controller
- The source code is very small but makes a serves as a good comparison to randomized testing
- The program is buggy: the abort statement in the program is reachable under certain program inputs
- First, to understand how this function was run you need to imagine that this function can be called an arbitrary number of times with different values for message
- It is essentially representing a state machine which causes transitions based on the input to the function
- The abort statement in the program can be reached after applying two messages: first passing 3 and then passing 0
- Because this bug takes at least two messages to manifest, the chance for a random tester to find it is one out of 2 to the sixty four, which is obviously very close to zero
- DART on the other hand, finds the bug in less than one second

AC-Controller

```c
int is_room_hot, ac, is_door_closed;
void ac_controller(int message) {
  if (message == 0) is_room_hot = 1;
  if (message == 1) is_room_hot = 0;
  if (message == 2) {
    is_door_closed = 0;
    ac = 0;
  }
  if (message == 3) {
    is_door_closed = 1;
    if (is_room_hot) ac = 1;
  }
  if (is_room_hot && is_door_closed
      && !ac) {
    abort();
  }
}
```

- Random testing does not work
- $2^{32} \times 2^{32} = 2^{64}$ number of possibilities
- One leads to the error

- First, we can look at the source code of the AC controller
- The source code is very small but makes a serves as a good comparison to randomized testing
- The program is buggy: the abort statement in the program is reachable under certain program inputs
- First, to understand how this function was run you need to imagine that this function can be called an arbitrary number of times with different values for message
- It is essentially representing a state machine which causes transitions based on the input to the function
- The abort statement in the program can be reached after applying two messages: first passing 3 and then passing 0
- Because this bug takes at least two messages to manifest, the chance for a random tester to find it is one out of 2 to the sixty four, which is obviously very close to zero
- DART on the other hand, finds the bug in less than one second

AC-Controller

```c
int is_room_hot, ac, is_door_closed;
void ac_controller(int message) {
  if (message == 0) is_room_hot = 1;
  if (message == 1) is_room_hot = 0;
  if (message == 2) {
    is_door_closed = 0;
    ac = 0;
  }
  if (message == 3) {
    is_door_closed = 1;
    if (is_room_hot) ac = 1;
  }
  if (is_room_hot && is_door_closed
      && !ac) {
    abort();
  }
}
```

- Random testing does not work
- $2^{32} \times 2^{32} = 2^{64}$ number of possibilities
- One leads to the error
- Never finds the bug after “hours”

- First, we can look at the source code of the AC controller
- The source code is very small but makes a serves as a good comparison to randomized testing
- The program is buggy: the abort statement in the program is reachable under certain program inputs
- First, to understand how this function was run you need to imagine that this function can be called an arbitrary number of times with different values for message
- It is essentially representing a state machine which causes transitions based on the input to the function
- The abort statement in the program can be reached after applying two messages: first passing 3 and then passing 0
- Because this bug takes at least two messages to manifest, the chance for a random tester to find it is one out of 2 to the sixty four, which is obviously very close to zero
- DART on the other hand, finds the bug in less than one second

AC-Controller

```c
int is_room_hot, ac, is_door_closed;
void ac_controller(int message) {
  if (message == 0) is_room_hot = 1;
  if (message == 1) is_room_hot = 0;
  if (message == 2) {
    is_door_closed = 0;
    ac = 0;
  }
  if (message == 3) {
    is_door_closed = 1;
    if (is_room_hot) ac = 1;
  }
  if (is_room_hot && is_door_closed
      && !ac) {
    abort();
  }
}
```

- Random testing does not work
- $2^{32} \times 2^{32} = 2^{64}$ number of possibilities
- One leads to the error
- Never finds the bug after "hours"
- DART: less than one second

- First, we can look at the source code of the AC controller
- The source code is very small but makes a serves as a good comparison to randomized testing
- The program is buggy: the abort statement in the program is reachable under certain program inputs
- First, to understand how this function was run you need to imagine that this function can be called an arbitrary number of times with different values for message
- It is essentially representing a state machine which causes transitions based on the input to the function
- The abort statement in the program can be reached after applying two messages: first passing 3 and then passing 0
- Because this bug takes at least two messages to manifest, the chance for a random tester to find it is one out of 2 to the sixty four, which is obviously very close to zero
- DART on the other hand, finds the bug in less than one second

Needham-Schroeder Protocol

- Protocol for two users to authenticate each other

- Next, the authors looked at the C implementation of the Needham-Schroeder protocol
- We do not need to consider the details of the protcol but is essentially a way for two users to start a secure communication channel
- The original algorithm contains a bug allowing an attacker to impersonate a user
- They tested on a 400 line C implementation
- They constrained the environment, or, the actions acceptable by the attacker to be as reasonable as the assumptions used in the paper describing the fault in the protocol
- Given these assumptions, DART was able to reproduce the fault in the protocol after 18 minutes of testing
- The author who originally reported the fault in the protocol proposed a fix
- Re running dart on the fixed protocol lead to another bug to be found which was acknowledged by the author
- It took DART 22 minutes to find this bug

Needham-Schroeder Protocol

- Protocol for two users to authenticate each other
- Contains impersonation bug

- Next, the authors looked at the C implementation of the Needham-Schroeder protocol
- We do not need to consider the details of the protcol but is essentially a way for two users to start a secure communication channel
- The original algorithm contains a bug allowing an attacker to impersonate a user
- They tested on a 400 line C implementation
- They constrained the environment, or, the actions acceptable by the attacker to be as reasonable as the assumptions used in the paper describing the fault in the protocol
- Given these assumptions, DART was able to reproduce the fault in the protocol after 18 minutes of testing
- The author who originally reported the fault in the protocol proposed a fix
- Re running dart on the fixed protocol lead to another bug to be found which was acknowledged by the author
- It took DART 22 minutes to find this bug

Needham-Schroeder Protocol

- Protocol for two users to authenticate each other
- Contains impersonation bug
- C implementation (400 LOC)

- Next, the authors looked at the C implementation of the Needham-Schroeder protocol
- We do not need to consider the details of the protcol but is essentially a way for two users to start a secure communication channel
- The original algorithm contains a bug allowing an attacker to impersonate a user
- They tested on a 400 line C implementation
- They constrained the environment, or, the actions acceptable by the attacker to be as reasonable as the assumptions used in the paper describing the fault in the protocol
- Given these assumptions, DART was able to reproduce the fault in the protocol after 18 minutes of testing
- The author who originally reported the fault in the protocol proposed a fix
- Re running dart on the fixed protocol lead to another bug to be found which was acknowledged by the author
- It took DART 22 minutes to find this bug

Needham-Schroeder Protocol

- Protocol for two users to authenticate each other
- Contains impersonation bug
- C implementation (400 LOC)
- Used “reasonable” environment constraints

- Next, the authors looked at the C implementation of the Needham-Schroeder protocol
- We do not need to consider the details of the protcol but is essentially a way for two users to start a secure communication channel
- The original algorithm contains a bug allowing an attacker to impersonate a user
- They tested on a 400 line C implementation
- They constrained the environment, or, the actions acceptable by the attacker to be as reasonable as the assumptions used in the paper describing the fault in the protocol
- Given these assumptions, DART was able to reproduce the fault in the protocol after 18 minutes of testing
- The author who originally reported the fault in the protocol proposed a fix
- Re running dart on the fixed protocol lead to another bug to be found which was acknowledged by the author
- It took DART 22 minutes to find this bug

Needham-Schroeder Protocol

- Protocol for two users to authenticate each other
- Contains impersonation bug
- C implementation (400 LOC)
- Used “reasonable” environment constraints
- Dart: 18 minutes to find error

- Next, the authors looked at the C implementation of the Needham-Schroeder protocol
- We do not need to consider the details of the protcol but is essentially a way for two users to start a secure communication channel
- The original algorithm contains a bug allowing an attacker to impersonate a user
- They tested on a 400 line C implementation
- They constrained the environment, or, the actions acceptable by the attacker to be as reasonable as the assumptions used in the paper describing the fault in the protocol
- Given these assumptions, DART was able to reproduce the fault in the protocol after 18 minutes of testing
- The author who originally reported the fault in the protocol proposed a fix
- Re running dart on the fixed protocol lead to another bug to be found which was acknowledged by the author
- It took DART 22 minutes to find this bug

Needham-Schroeder Protocol

- Protocol for two users to authenticate each other
- Contains impersonation bug
- C implementation (400 LOC)
- Used "reasonable" environment constraints
- Dart: 18 minutes to find error
- Re-ran on "fixed" version: found another bug

- Next, the authors looked at the C implementation of the Needham-Schroeder protocol
- We do not need to consider the details of the protcol but is essentially a way for two users to start a secure communication channel
- The original algorithm contains a bug allowing an attacker to impersonate a user
- They tested on a 400 line C implementation
- They constrained the environment, or, the actions acceptable by the attacker to be as reasonable as the assumptions used in the paper describing the fault in the protocol
- Given these assumptions, DART was able to reproduce the fault in the protocol after 18 minutes of testing
- The author who originally reported the fault in the protocol proposed a fix
- Re running dart on the fixed protocol lead to another bug to be found which was acknowledged by the author
- It took DART 22 minutes to find this bug

Needham-Schroeder Protocol

- Protocol for two users to authenticate each other
- Contains impersonation bug
- C implementation (400 LOC)
- Used "reasonable" environment constraints
- Dart: 18 minutes to find error
- Re-ran on "fixed" version: found another bug
  - 22 minutes

- Next, the authors looked at the C implementation of the Needham-Schroeder protocol
- We do not need to consider the details of the protcol but is essentially a way for two users to start a secure communication channel
- The original algorithm contains a bug allowing an attacker to impersonate a user
- They tested on a 400 line C implementation
- They constrained the environment, or, the actions acceptable by the attacker to be as reasonable as the assumptions used in the paper describing the fault in the protocol
- Given these assumptions, DART was able to reproduce the fault in the protocol after 18 minutes of testing
- The author who originally reported the fault in the protocol proposed a fix
- Re running dart on the fixed protocol lead to another bug to be found which was acknowledged by the author
- It took DART 22 minutes to find this bug

oSIP

- oSIP: Telephone over IP library

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

oSIP

- oSIP: Telephone over IP library
- Tested external functions

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

oSIP

- oSIP: Telephone over IP library
- Tested external functions
- Found many functions not checking NULL pointers

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

oSIP

- oSIP: Telephone over IP library
- Tested external functions
- Found many functions not checking NULL pointers
- Found denial of service in parser

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

oSIP

- oSIP: Telephone over IP library
- Tested external functions
- Found many functions not checking NULL pointers
- Found denial of service in parser
  - Request too large a stack frame

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

oSIP

- oSIP: Telephone over IP library
- Tested external functions
- Found many functions not checking NULL pointers
- Found denial of service in parser
  - Request too large a stack frame
  - Return of `alloca` not checked

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

oSIP

- oSIP: Telephone over IP library
- Tested external functions
- Found many functions not checking NULL pointers
- Found denial of service in parser
  - Request too large a stack frame
  - Return of `alloca` not checked
- "Bugs" fixed by developers

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

oSIP

- oSIP: Telephone over IP library
- Tested external functions
- Found many functions not checking NULL pointers
- Found denial of service in parser
  - Request too large a stack frame
  - Return of `alloca` not checked
- "Bugs" fixed by developers
- Intuition: specifications make this technique much better

- oSIP is essentially a library implementing telephone and other multi-media stuff over IP
- The authors tested the external library functions
- First, they found many functions which crash when passed a NULL pointer because the function seemed to assume the pointers were non-null
- The authors moved onto looking at more functions in the program and found a potential way to crash the library
- The crash involved an input allocating too much space on the stack; the library does not check the return of the alloca call, which could be NULL, causing a crash
- Because there is not a clear specification, the authors were not sure if these were real bugs, but they note that the parser issue was fixed by the developers
- Though the authors do not mention it, this points at one of the issues of making a practical directed testing framework which is that the tool produces more meaningful results if there is a specification present

Overview

Introduction

Path Constraints

Experimental Results

Conclusions and Questions

Next, I'll go over some conclusions and open questions in the paper

Open Questions

- How to handle concurrent programs?

- The paper leaves some questions open at the time of writing
- First, the authors are only considering branches as a source of non-determinism in the program
- In the case of a concurrent program, it is not clear how the technique could simultaneously generate inputs to check both the branches and thread schedules
- There was, however, an interesting sounding paper by some cool authors in this years FSE extending the DART approach to efficiently handle multi-threaded programs
- Second, the analysis is bounded: its not clear how or if a technique such as this can be used in an unbounded analysis
- And third, it is not too clear how scalable this analysis is
- For example, if there are very complicated functions or those using very long loops or recurions, its not clear if the constraints generated by the analysis will be solvable

Open Questions

- How to handle concurrent programs?
  - Branches and thread schedules?

- The paper leaves some questions open at the time of writing
- First, the authors are only considering branches as a source of non-determinism in the program
- In the case of a concurrent program, it is not clear how the technique could simultaneously generate inputs to check both the branches and thread schedules
- There was, however, an interesting sounding paper by some cool authors in this years FSE extending the DART approach to efficiently handle multi-threaded programs
- Second, the analysis is bounded: its not clear how or if a technique such as this can be used in an unbounded analysis
- And third, it is not too clear how scalable this analysis is
- For example, if there are very complicated functions or those using very long loops or recurions, its not clear if the constraints generated by the analysis will be solvable

Open Questions

- How to handle concurrent programs?
  - Branches and thread schedules?
  - *Assertion Guided Symbolic Execution of Multithreaded Programs*, Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, Aarti Gupta. FSE '15

- The paper leaves some questions open at the time of writing
- First, the authors are only considering branches as a source of non-determinism in the program
- In the case of a concurrent program, it is not clear how the technique could simultaneously generate inputs to check both the branches and thread schedules
- There was, however, an interesting sounding paper by some cool authors in this years FSE extending the DART approach to efficiently handle multi-threaded programs
- Second, the analysis is bounded: its not clear how or if a technique such as this can be used in an unbounded analysis
- And third, it is not too clear how scalable this analysis is
- For example, if there are very complicated functions or those using very long loops or recurions, its not clear if the constraints generated by the analysis will be solvable

Open Questions

- How to handle concurrent programs?
  - Branches and thread schedules?
  - *Assertion Guided Symbolic Execution of Multithreaded Programs*, Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, Aarti Gupta. FSE '15
- How to handle unbounded programs?

- The paper leaves some questions open at the time of writing
- First, the authors are only considering branches as a source of non-determinism in the program
- In the case of a concurrent program, it is not clear how the technique could simultaneously generate inputs to check both the branches and thread schedules
- There was, however, an interesting sounding paper by some cool authors in this years FSE extending the DART approach to efficiently handle multi-threaded programs
- Second, the analysis is bounded: its not clear how or if a technique such as this can be used in an unbounded analysis
- And third, it is not too clear how scalable this analysis is
- For example, if there are very complicated functions or those using very long loops or recurions, its not clear if the constraints generated by the analysis will be solvable

Open Questions

- How to handle concurrent programs?
  - Branches and thread schedules?
  - *Assertion Guided Symbolic Execution of Multithreaded Programs*, Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, Aarti Gupta. FSE '15
- How to handle unbounded programs?
- How scalable is this approach?

- The paper leaves some questions open at the time of writing
- First, the authors are only considering branches as a source of non-determinism in the program
- In the case of a concurrent program, it is not clear how the technique could simultaneously generate inputs to check both the branches and thread schedules
- There was, however, an interesting sounding paper by some cool authors in this years FSE extending the DART approach to efficiently handle multi-threaded programs
- Second, the analysis is bounded: its not clear how or if a technique such as this can be used in an unbounded analysis
- And third, it is not too clear how scalable this analysis is
- For example, if there are very complicated functions or those using very long loops or recurions, its not clear if the constraints generated by the analysis will be solvable

Conclusion

- Function-test generation

- So, in conclusion I presented DART, a tool to generate test inputs for functions in order to automated the creation of unit tests
- The technique is fully automated in that the developer does not need to hand generate test inputs to exercise new paths in a function
- The experimental results showed that the technique is faster than simple random testing
- With that, I'll take any questions

Conclusion

- Function-test generation
- Fully automated

- So, in conclusion I presented DART, a tool to generate test inputs for functions in order to automated the creation of unit tests
- The technique is fully automated in that the developer does not need to hand generate test inputs to exercise new paths in a function
- The experimental results showed that the technique is faster than simple random testing
- With that, I'll take any questions

Conclusion

- Function-test generation
- Fully automated
- Faster than random testing

- So, in conclusion I presented DART, a tool to generate test inputs for functions in order to automated the creation of unit tests
- The technique is fully automated in that the developer does not need to hand generate test inputs to exercise new paths in a function
- The experimental results showed that the technique is faster than simple random testing
- With that, I'll take any questions

Conclusion

- Function-test generation
- Fully automated
- Faster than random testing

- So, in conclusion I presented DART, a tool to generate test inputs for functions in order to automated the creation of unit tests
- The technique is fully automated in that the developer does not need to hand generate test inputs to exercise new paths in a function
- The experimental results showed that the technique is faster than simple random testing
- With that, I'll take any questions

Conclusion

- Function-test generation
- Fully automated
- Faster than random testing

Questions?

- So, in conclusion I presented DART, a tool to generate test inputs for functions in order to automated the creation of unit tests
- The technique is fully automated in that the developer does not need to hand generate test inputs to exercise new paths in a function
- The experimental results showed that the technique is faster than simple random testing
- With that, I'll take any questions