



Automating Structural Testing of C Programs: Experience with PathCrawler

Bernard Botella, *Mickaël Delahaye*,
Stéphane Hong-Tuan-Ha, Nikolai Kosmatov,
Patricia Mouy, Muriel Roger and Nicky Williams

CEA LIST, Software Reliability Laboratory

May 18, 2009

Outline



Generation method and tool

Combining symbolic and concrete executions
Advantages and positioning

Obstacles to industrial application

Path Explosion
Library function calls
Test context
And more

Outline



Generation method and tool

Combining symbolic and concrete executions
Advantages and positioning

Obstacles to industrial application

Path Explosion
Library function calls
Test context
And more



- ▶ Generates test cases from C source code (also C++)
- ▶ Path-based method which combines symbolic and concrete executions
- ▶ Uses Constraint Logic Programming to find test inputs
 - ▶ Colibri, a constraint solver shared by several CEA tools (GATEL, OSMOSE)
- ▶ Applied on real industrial examples of embedded software

Combining *Symbolic* and *Concrete*



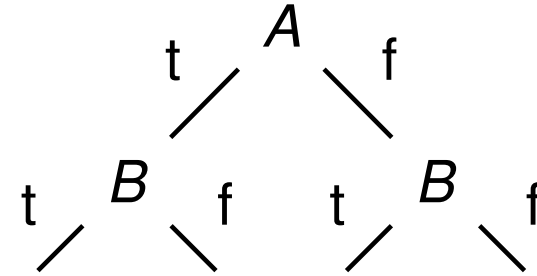
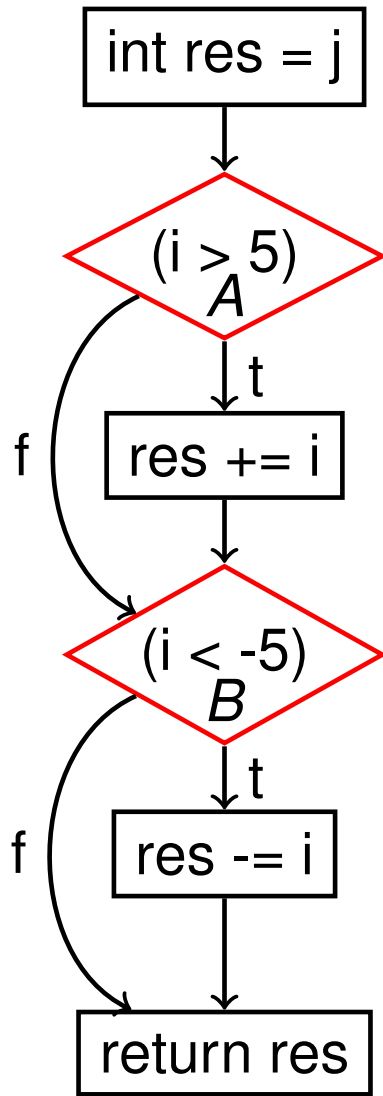
PC (Williams et al. 2004), DART (Godefroid et al. 2005)
a.k.a *concolic execution* or *dynamic symbolic execution*

- ▶ Symbolic execution computes *path condition* on a path prefix.
- ▶ If there is a solution to the path prefix, this solution is a test input.
- ▶ The concrete execution of the program on this input gives us a full trace/path thanks to instrumentation.
- ▶ The obtained path necessarily starts with the path prefix.

Example



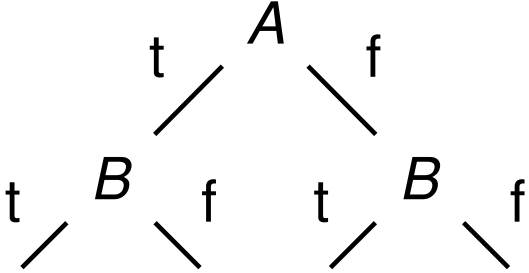
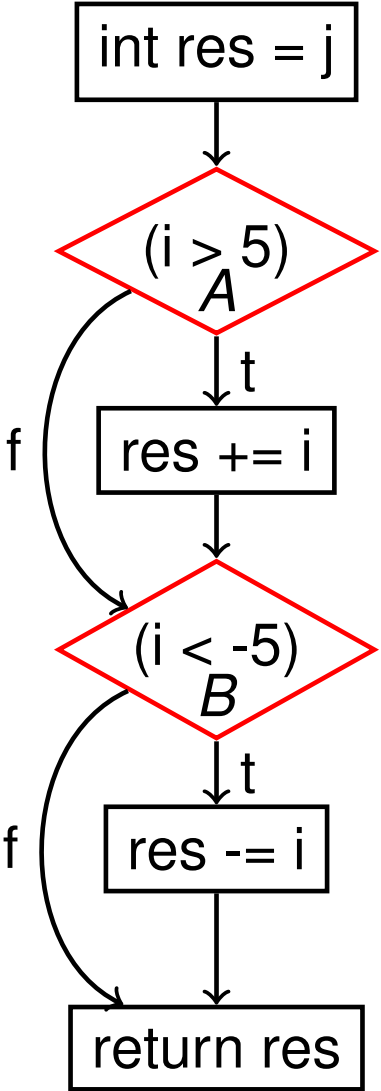
int f(int i, int j)



Example



```
int f(int i, int j)
```

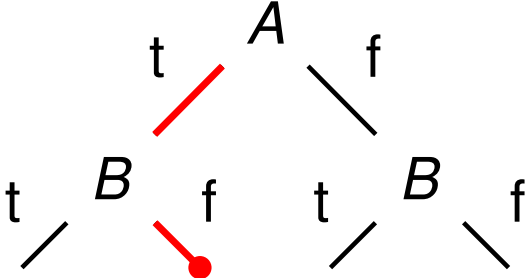
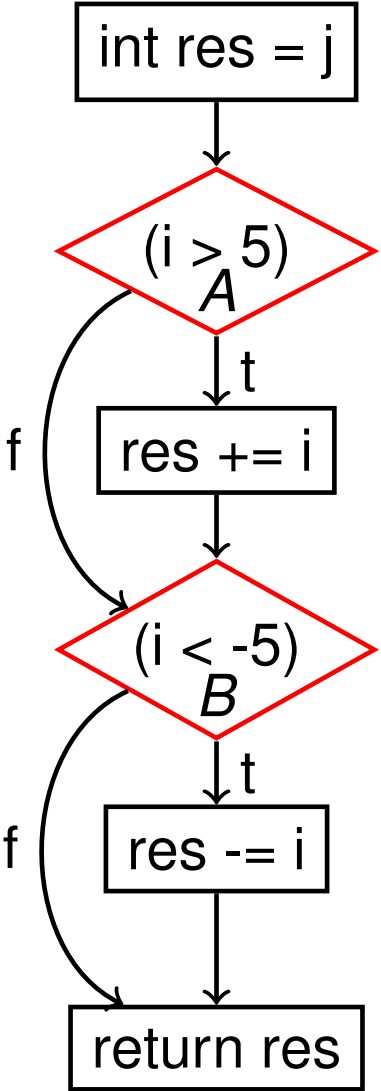


Starts by choosing an arbitrary valid input, say $i=24$ and $j=-50$
Concretely executes the function on this input

Example



```
int f(int i, int j)
```



The concrete execution allows us to know the path activated by $i=24$ and $j=-50$

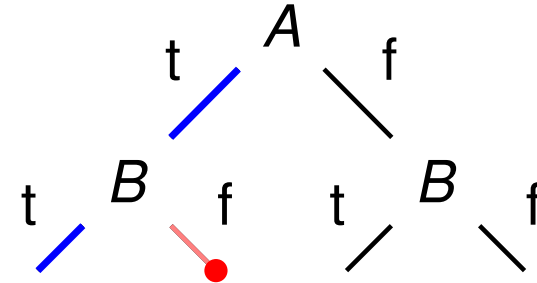
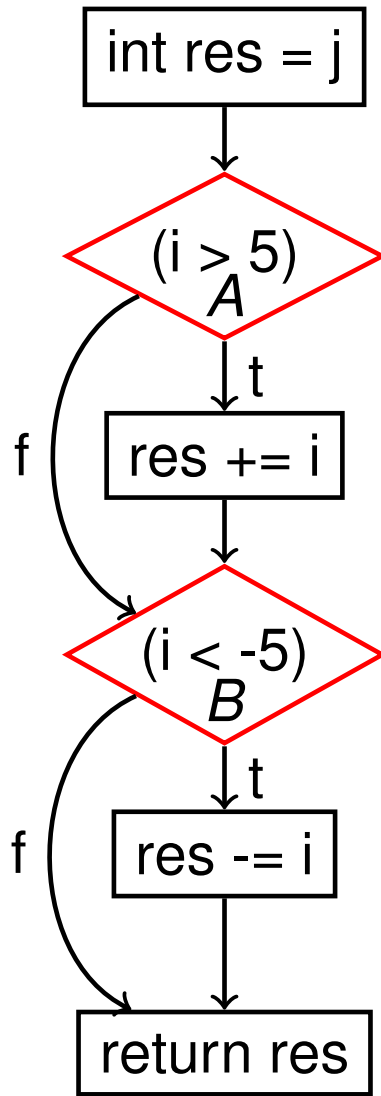
Test cases and coverage:

- ▶ $i=24, j=-50$ $A^t B^f$

Example



int f(int i, int j)



Depth-first strategy:

next path to try is $A^t B^t$

Uses symbolic execution to compute the *path condition*: $i > 5 \wedge i < -5$

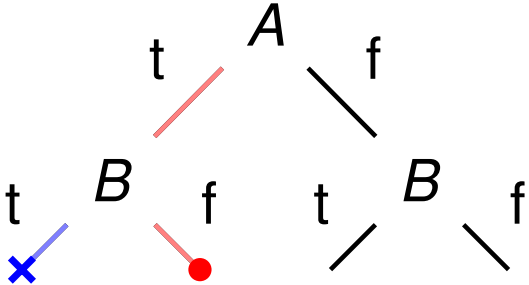
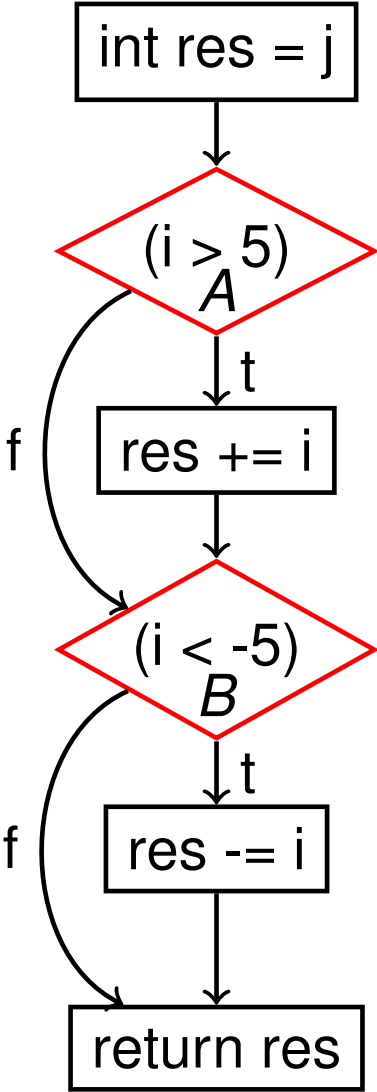
Test cases and coverage:

- ▶ $i=24, j=-50$ $A^t B^f$

Example



```
int f(int i, int j)
```



The solver finds $i > 5 \wedge i < -5$ inconsistent: the path is infeasible.

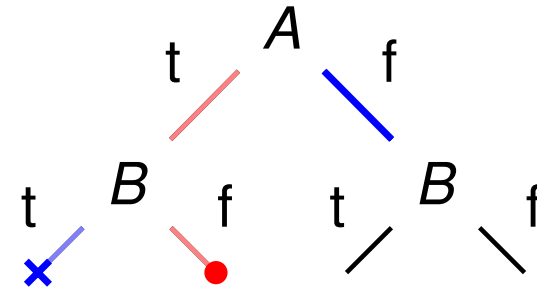
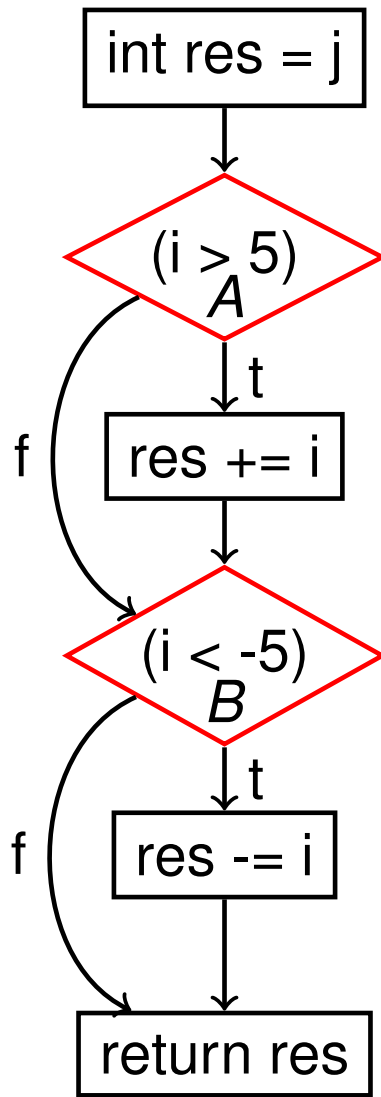
Test cases and coverage:

- ▶ $i=24, j=-50 A^t B^f$

Example



int f(int i, int j)



Continues the DFS and symbolically executes A^f

Finds a solution to the path condition $i \leq 5$, say $i=-72$ and $j=-63$

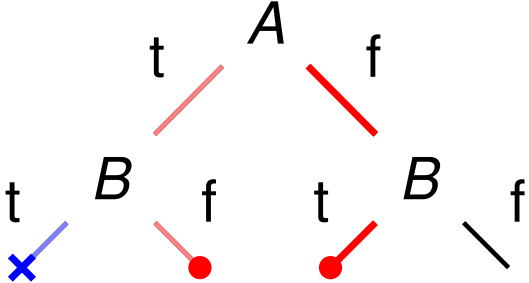
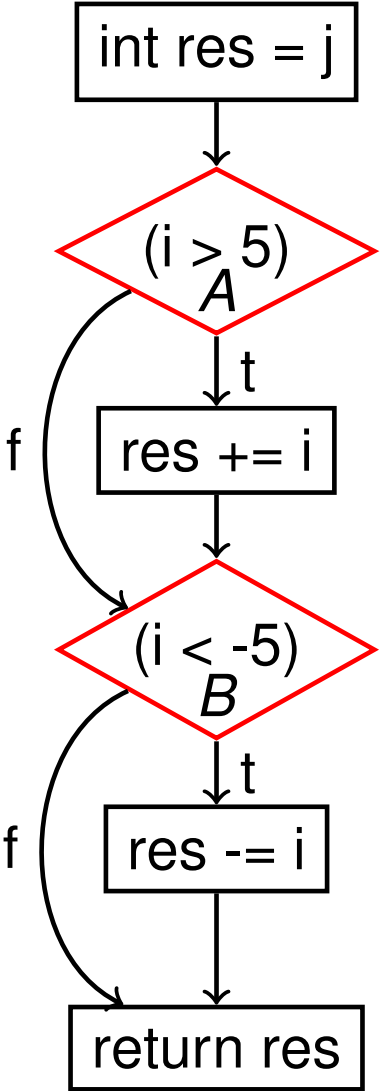
Test cases and coverage:

- ▶ $i=24, j=-50 A^t B^f$

Example



```
int f(int i, int j)
```



Runs a concrete execution to find the followed path

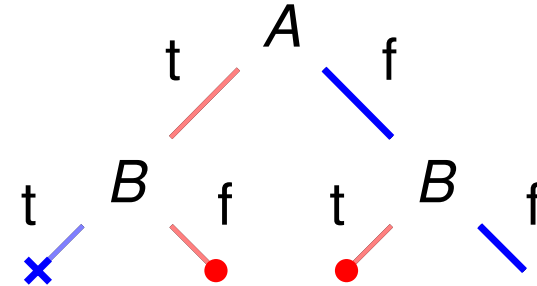
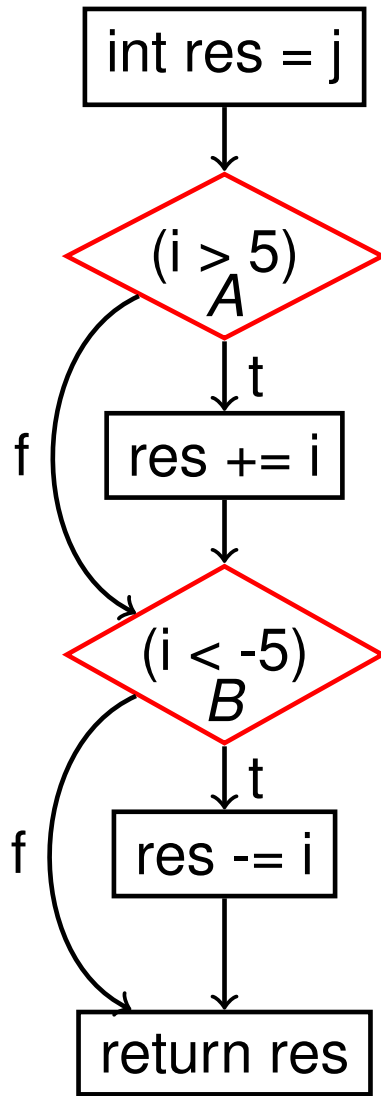
Test cases and coverage:

- ▶ $i=24, j=-50 A^t B^f$
- ▶ $i=-72, j=-63 A^f B^t$

Example



int f(int i, int j)



Then tries the last prefix $A^f B^f$:
 $-5 \leq i \leq 5 \rightsquigarrow i=1$ and $j=87$.

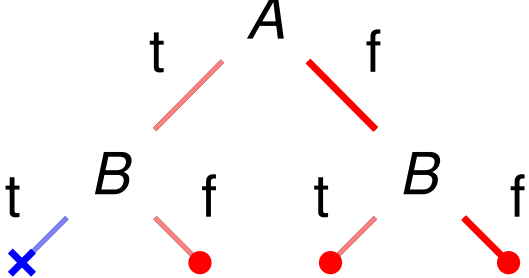
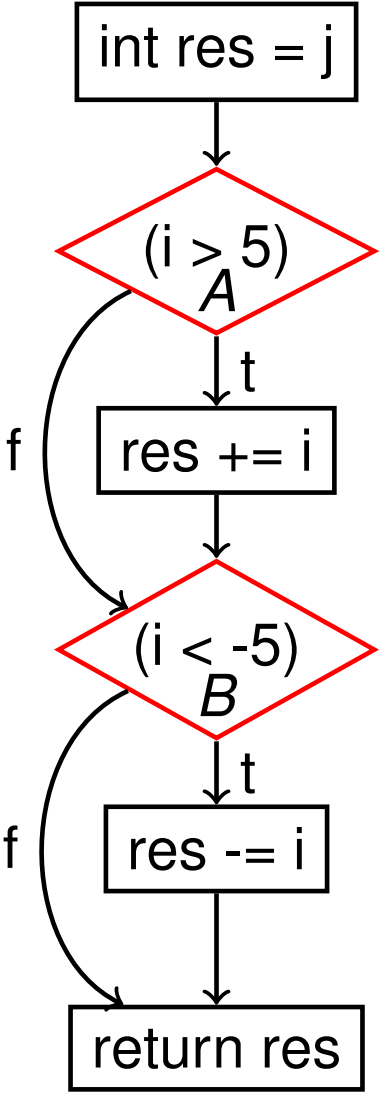
Test cases and coverage:

- ▶ $i=24, j=-50 A^t B^f$
- ▶ $i=-72, j=-63 A^f B^t$

Example



```
int f(int i, int j)
```



Runs a concrete execution to confirm the activated path

Test cases and coverage:

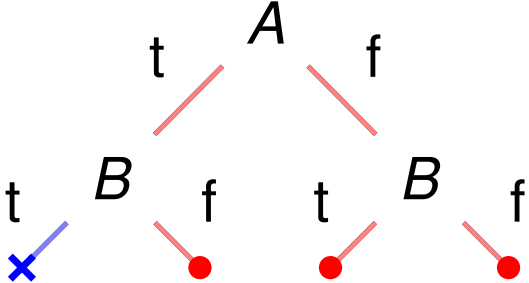
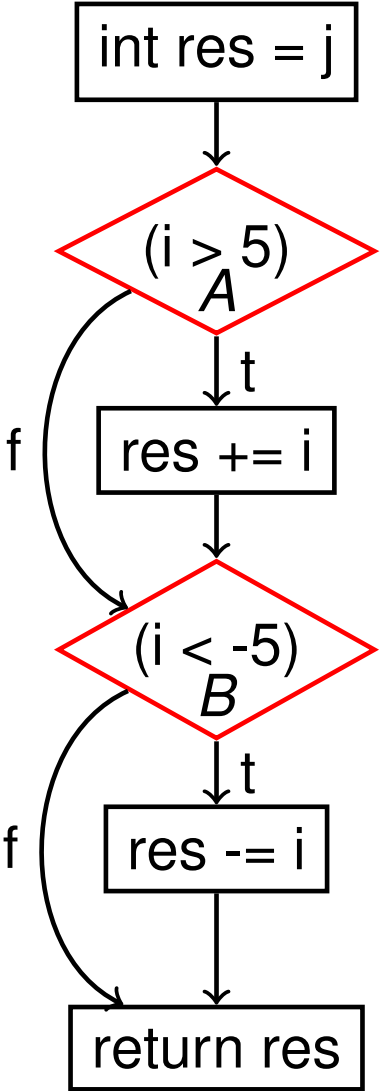
- ▶ $i=24, j=-50 A^t B^f$
- ▶ $i=-72, j=-63 A^f B^t$
- ▶ $i=1, j=87 A^t B^t$



Example



```
int f(int i, int j)
```



3 test cases found covering all feasible paths

Test cases and coverage:

- ▶ $i=24, j=-50 A^t B^f$
- ▶ $i=-72, j=-63 A^f B^t$
- ▶ $i=1, j=87 A^t B^t$



Advantages



Sound Concrete execution of the test data ensures the test objective is actually reached.

Complete If the solver does not reach its time limit, all feasible paths (given the criteria) are covered.

Incremental The depth-first strategy is incremental. Particularly useful for constraint propagation.

Fast entry From a path condition on a (possibly empty) path prefix, it gets to cover a full path

Positioning



Thou shalt not leave a path uncovered!

PathCrawler Team

PathCrawler is designed to search for maximum coverage of critical code. The objective is not to detect bug but to justify a certain degree of quality.

Strong coverage criteria

All-paths No limits whatsoever. Too many tests in the general case.

All-k-paths Bounds the number of consecutive loop iterations.

Outline



Generation method and tool

Combining symbolic and concrete executions
Advantages and positioning

Obstacles to industrial application

Path Explosion
Library function calls
Test context
And more

Issue 1: Path Explosion



A program may contain lots of potential paths:

- ▶ A function containing a sequence of n if-then-else statements has at least 2^n potential paths
- ▶ Loop bounds may depend on the input
- ▶ Inlining function calls worsens the problem

Some approaches:

- ▶ Relax the coverage criterion
- ▶ Change the strategy to improve the coverage curve over time (EXE, Pex)
- ▶ Limit the path dimension:
 - ▶ k -paths, n -paths
- ▶ Prevent redundant testing:
 - ▶ State caching (RWSet)
 - ▶ Compositional test generation (SMART)

Issue 2: Library function calls



- ▶ Real programs often call functions whose source code is not available.
- ▶ Even if we have the code we might be completely uninterested in the path in some called functions unless it impacts the calling function.

Issue 2: Library function calls



- ▶ Real programs often call functions whose source code is not available.
- ▶ Even if we have the code we might be completely uninterested in the path in some called functions unless it impacts the calling function.
- ▶ Ad-hoc stubs
- ▶ Use concrete values as expected return values

Issue 2: Library function calls



- ▶ Real programs often call functions whose source code is not available.
- ▶ Even if we have the code we might be completely uninterested in the path in some called functions unless it impacts the calling function.
- ▶ Ad-hoc stubs
- ▶ Use concrete values as expected return values
- ▶ *Grey box testing: use library function specifications*

PathCrawler (Mouy et al. 2008) uses pre-/post-condition pairs to abstract library function calls and offers two variations of the coverage criteria:

- ▶ all-(k)-paths and all functional domains
- ▶ all-(k)-paths with lazy exploration of functional domains



Issue 3: Test context



Functions come with formal or informal conditions on the input:

- ▶ The definition domain, e.g. `sqrt` is only defined for positive numbers. This gives us a first test context.
- ▶ The user may impose additional restrictions on the context.

Testing with no context can be expensive and distractive:

- ▶ No robustness gain if the intended programming style is not defensive.
- ▶ Loses time detecting bugs for unrealistic inputs.
- ▶ Any additional restriction cut some paths.

Issue 3: Expression of the test context



Here we call the conditions of the inputs for which a function is to be tested the *precondition* of the function.

$$\text{testdom}(f) = \{x \in t_1 \times \dots \times t_n \mid \text{Pre}(f, x)\}$$

Issue 3: Expression of the test context



Here we call the conditions of the inputs for which a function is to be tested the *precondition* of the function.

$$\text{testdom}(f) = \{x \in t_1 \times \dots \times t_n \mid \text{Pre}(f, x)\}$$

Specifying $\text{Pre}(f, x)$ can be difficult especially for imperative language developers...

- ▶ Expression in constraints:
powerful and efficient *but unnatural for developers*
- ▶ Directly in the same language as the program under test

Issue 3: From C precondition to constraints?



For instance, any binary search implementation needs the array to be sorted.

Here is a C function coding this precondition:

```
int bsearch_precond(int arr[4], int key) {  
    for (int i=1; i < 4; i++)  
        if (arr[4] < arr[i-1]) return 0;  
    return 1;  
}
```

PathCrawler uses dynamic symbolic execution to find test inputs for which the C precondition returns 1.

Issue 3: Test context in other tools



- ▶ consistency check for structure invariants (JPF)
- ▶ construction or “*finitization*” (BORAT): rather describe how to construct than how to check consistency
- ▶ a bit of both (e.g. CUTE or PEX)
 - ▶ construction allows to overcome the limits of generalized symbolic execution
 - ▶ assumptions allow to specify properties on the test input

Challenge

How to find valid inputs “lazily”, i.e. without exercising all paths in the test context?

And more



- ▶ Memory model and aliasing
C language allows complex memory operations: arrays, pointers and pointer arithmetics
- ▶ Floating point numbers are not real numbers
Handling fpn as real may lead to discrepancy between symbolic and concolic executions
- ▶ Object oriented features (C++)
- ▶ Cyclical reactive and multitasking software

Conclusion



Experience tells us automated path-based test generation is close to be applicable to industrial programs.

We identified:

- ▶ major obstacles to industrial application
- ▶ evaluation criteria for automated software testing tool
- ▶ research directions

Conclusion



Experience tells us automated path-based test generation is close to be applicable to industrial programs.

We identified:

- ▶ major obstacles to industrial application
- ▶ evaluation criteria for automated software testing tool
- ▶ research directions

Thank you all for your attention!
If you have any question...

Outline



Memory model

A complex memory



The C language allows: arrays, bitwise operations, pointers (unmanaged references), pointer arithmetic and full reinterpretation (type casting on pointers).

PathCrawler's current choices

- ▶ constructs the input lazily (creating tree-like structure)
- ▶ always chooses a valid access if any
- ▶ does not allow to reinterpret memory chunks

However, it remains a major difficulty: memory aliasing, ie there may be multiple way to access the same data location.

Memory aliasing



```
int n = 1 ; int* p = &n;
```

n and $*p$ address the same memory location: they are aliases.

The difficulty arises when unknown inputs are used directly or indirectly in an array index or a pointer offset.

```
int t[] = {1, 2, 3};
```

Given an input i , $t[i]$ may be $t[0]$, $t[1]$ and $t[2]$

Multiple dereferencing: $*(*p + i) + j$

Whereas most other tools (CUTE, EXE and PEX) choose to fix all variable offsets except one, PathCrawler keeps tracks of these so as not to lose any feasible path due to under-approximation.