COMP 737011 - Memory Safety and Programming Language Design

Lecture 12: Dynamic Analysis of Rust Code

徐 辉
xuh@fudan.edu.cn

Outline

- 1. Background of Dynamic Analysis
- 2. Blackbox Fuzzing
- 3. Fuzz Target Generation
- 4. Dynamic Symbolic Execution

1. Background of Dynamic Analysis

Any Bugs in genvec()?

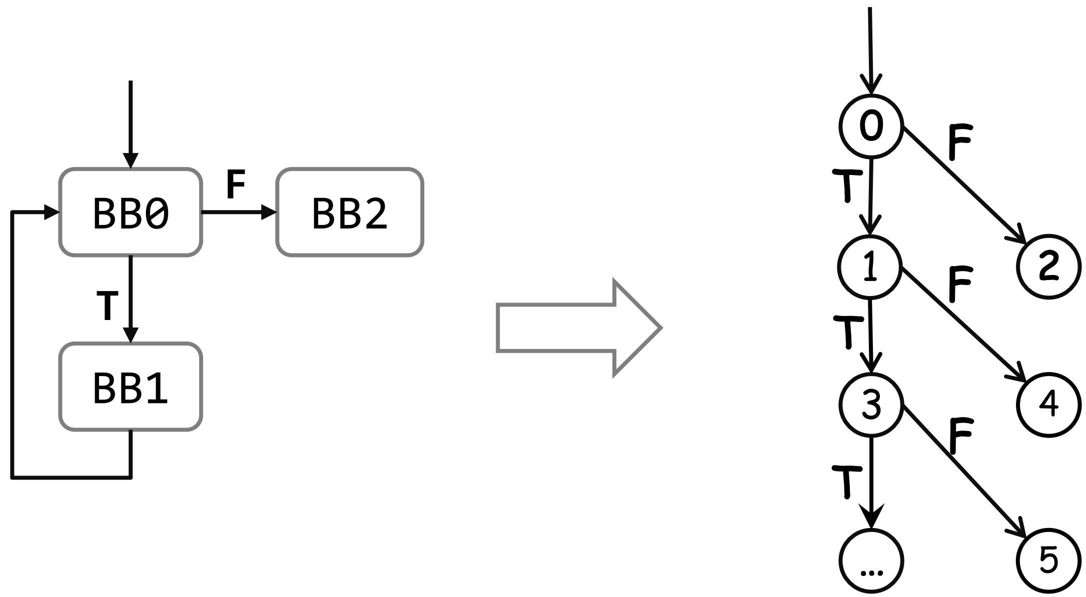
```
fn genvec(s:&mut String)->Vec<u8>{
    let mut v = Vec::new();
    let l = s.len();
    if l%10 == 0 {
        let ptr = s.as_mut_ptr();
        unsafe{
            v = Vec::from_raw_parts(ptr,l,s.len());
        }
    }
    return v;
}

fn main(){
    let args:Vec<String> = env::args().collect();
    let mut s = String::from(&args[1]);
    let v = genvec(&mut s);
    println!("{:?}",v);
}
```

Challenge of Bug Detection

- Testing?
  - Bugs are on uncommon execution paths.
  - How to achieve high coverage (code/path)
    - define test cases mannually?
    - automated test generation
  - Executing all paths is almost infeasible
- How to test Library APIs?

Programs as Computation Trees

- A control-flow graph with loops unrolled.
- Each node is a a conditional statement (state)
- Each edge is a sequence of sequential statements
- Each path represents an equivalence class of inputs.

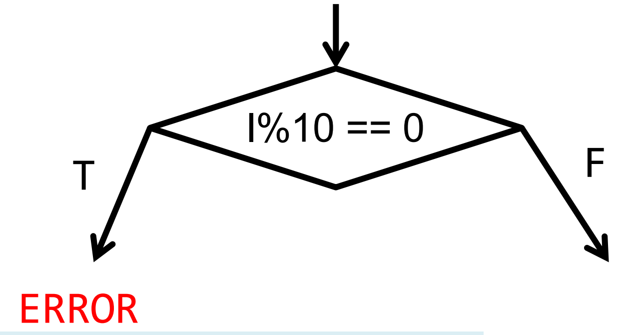


Control-flow graph

Computation tree

Example of Computation Tree

```
fn genvec(s:&mut String)->Vec<u8>{
    let mut v = Vec::new();
    let l = s.len();
    if l%10 == 0 {
        let ptr = s.as_mut_ptr();
        unsafe{
            v = Vec::from_raw_parts(ptr,l,s.len());
        }
    }
    return v;
}
```



How to Traverse the Tree?

- Random testing or fuzzing:
  - generate random inputs to execute the program.
  - probability of reaching error could be very small
- Symbolic execution:
  - Collect the constraints for each path
  - Use constraint solvers to find solutions

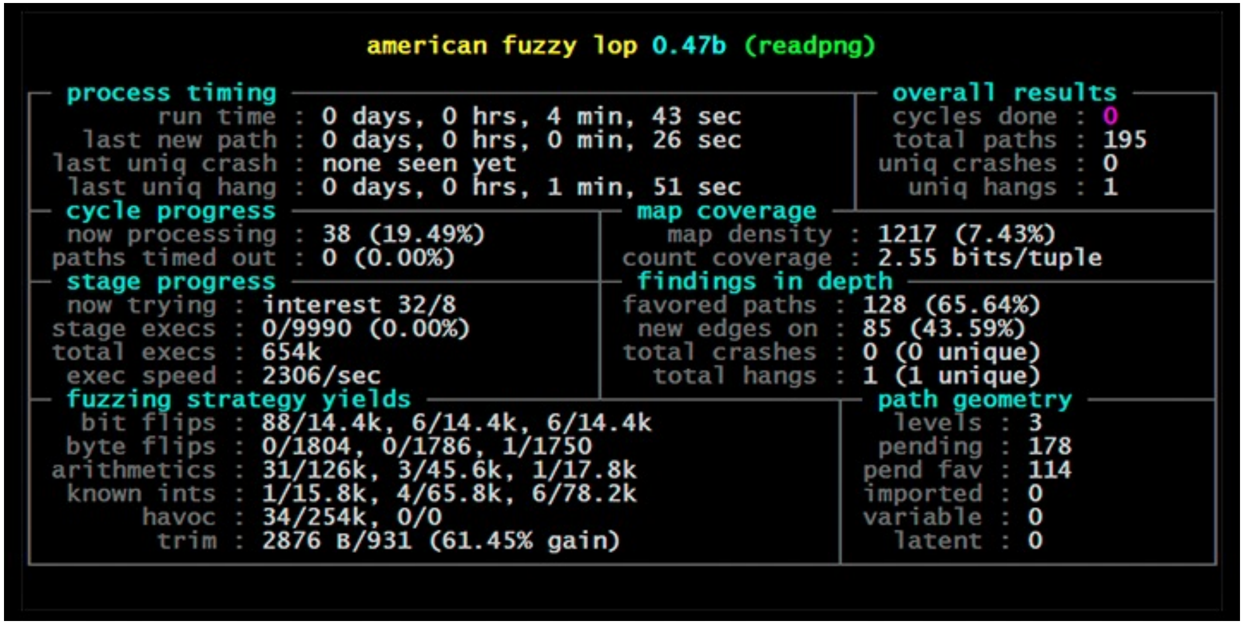
```
if x == 94389 {
    unreachable!();
}
```

Probability = $1/2^{32}$

2. Blackbox Fuzzing

Evolutionary Fuzzing with AFL

- AFL is the most famous fuzzing tool
  - http://lcamtuf.coredump.cx/afl/
  - Many followup tools available, e.g., afl++
- Use a genetic approach to learn interesting mutations:
  - trigger code code => keep it
  - cannot trigger new code => discard



Fuzz Rust with AFL

1. Install AFL

https://rust-fuzz.github.io/book/introduction.html

```
#: cargo install afl
```

2. Create a new project

```
#: cargo new fuzztarget
```

3. Add deps in Cargo.toml

```
[dependencies] afl = "*"
```

4. fuzz target

```
#[macro_use]
extern crate afl;
fn genvec(s:&mut String)->Vec<u8>{ ...}
fn main(){
    fuzz!(|data: &[u8]| {
        let mut s = String::from(&data);
        let v = genvec(&mut s);
    });
}
```

5. build the fuzz target

```
#: cargo afl build
```

6. add initial test seeds into a folder and then fuzz

```
#: cargo afl fuzz -i initcases -o output targetbin
```

Overall Algorithm

Init: load user-supplied initial test cases into the queue

Load: take next input file from the queue

Trim: attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program

Mutate: repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies

Score: if any of the mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue

The discovered test cases are also periodically culled to eliminate ones that have been obsoleted by newer, higher-coverage finds

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

Instrumentation

- Support programs
  - written in C/C++/Objective C
  - variants for Python/Go/Rust/OCaml
- Code instrumented to observe execution paths:
  - if source code is available, using modified compiler
  - if source code is unavailable, running code in an emulator
- Code coverage represented as a 64KB bitmap
  - different executions may collide with a small chance.
  - tradeoff between collision and efficiency

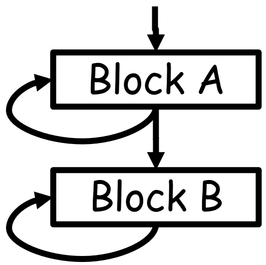
How to Instrument the Source Code?

- Objective: Different jumps from src to dest should result in different offsets of the bitmap.
- Where to inject?
  - at every branching point
- What code to inject?
  - use a random number as the id/hash of the block.
  - update the corresponding bitmap cell

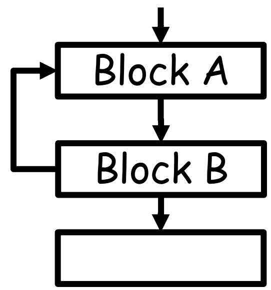
```
Psudocode:
cur_location = <COMPILE_TIME_RANDOM_FOR_THE_BLOCK>;
bitmap[cur_location ⊕ (prev_location >> 1)]++;
```

More about the Design

- Why shift 1 bit? Consider the following cases



How to differentiate
- Block A -> Block A
- Block B -> Block B



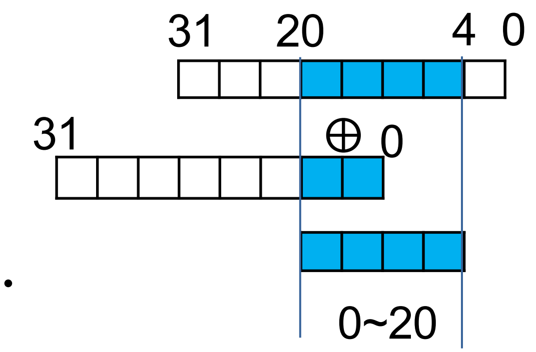
How to differentiate
- Block A -> Block B
- Block B -> Block A

- Edge coverage in nature
- Collision rate?

Branch cnt	Colliding tuples	Example targets
1,000	0.75%	giflib, lzo
2,000	1.5%	zlib, tar, xz
5,000	3.5%	libpng, libwebp
10,000	7%	libxml
20,000	14%	sqlite
50,000	30%	-

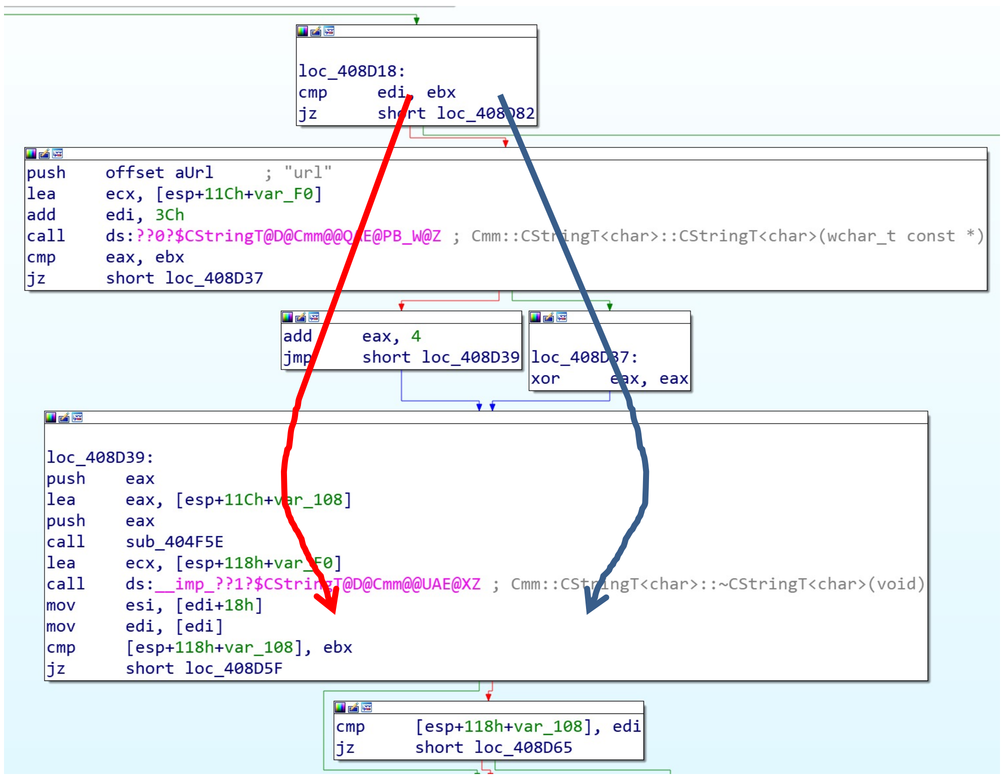
How to Instrument the Binaries

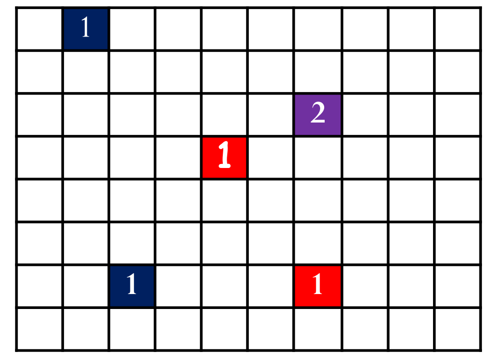
- Use the instruction address?
  - 32/64 bit
  - The bitmap offset is only 16bit;
- Select important bits heuristically.




```
Psudocode:
cur_location = (block_addr >> 4) ⊕ (block_addr << 8);
bitmap[cur_location ⊕ (prev_location >> 1)]++;
```

Visualize Bitmap





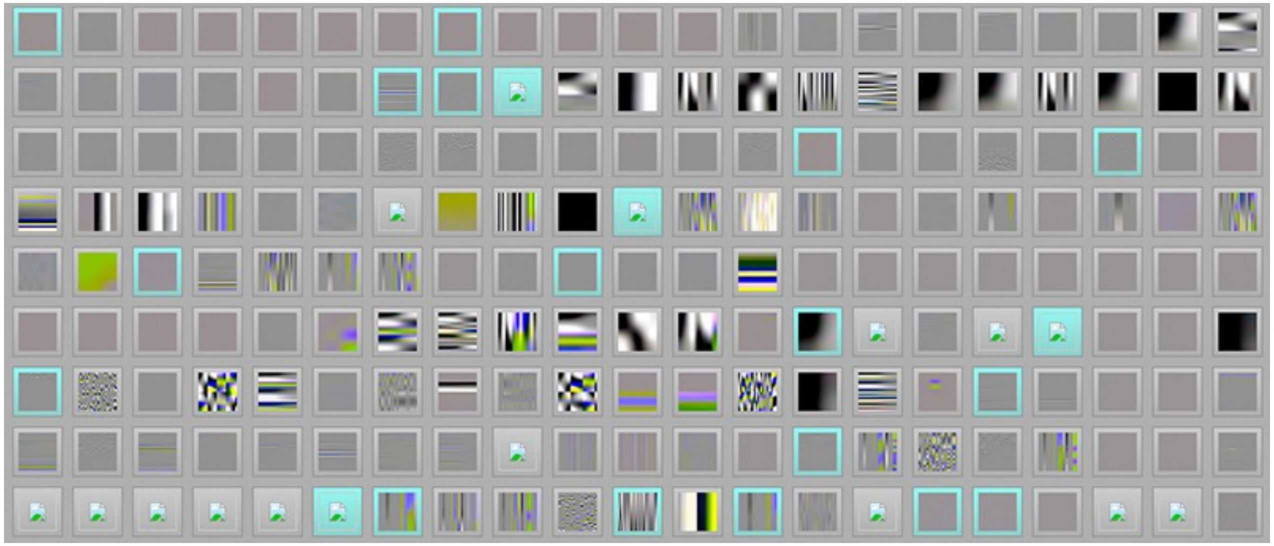
Bitmap (64KB)

Sample Mutation Strategies

- Bit flip
- Increase/decrease integers
- Use predefined interesting values
  - *eg.*, 0, -1, MAX_INT for integer
- Delete/combine/zero input block

Example: Learning the JPG file format

- Task: to fuzz a program that expects a JPG as input
- Experiment: start with 'hello world' as initial test input
- Result:
  - Generate the first image after six hours on an 8-core system
  - Produced many interesting pics by using the image as a seed



https://lcamtuf.blogspot.it/2014/11/pulling-jpegs-out-of-thin-air.html

3. Fuzz Target Generation

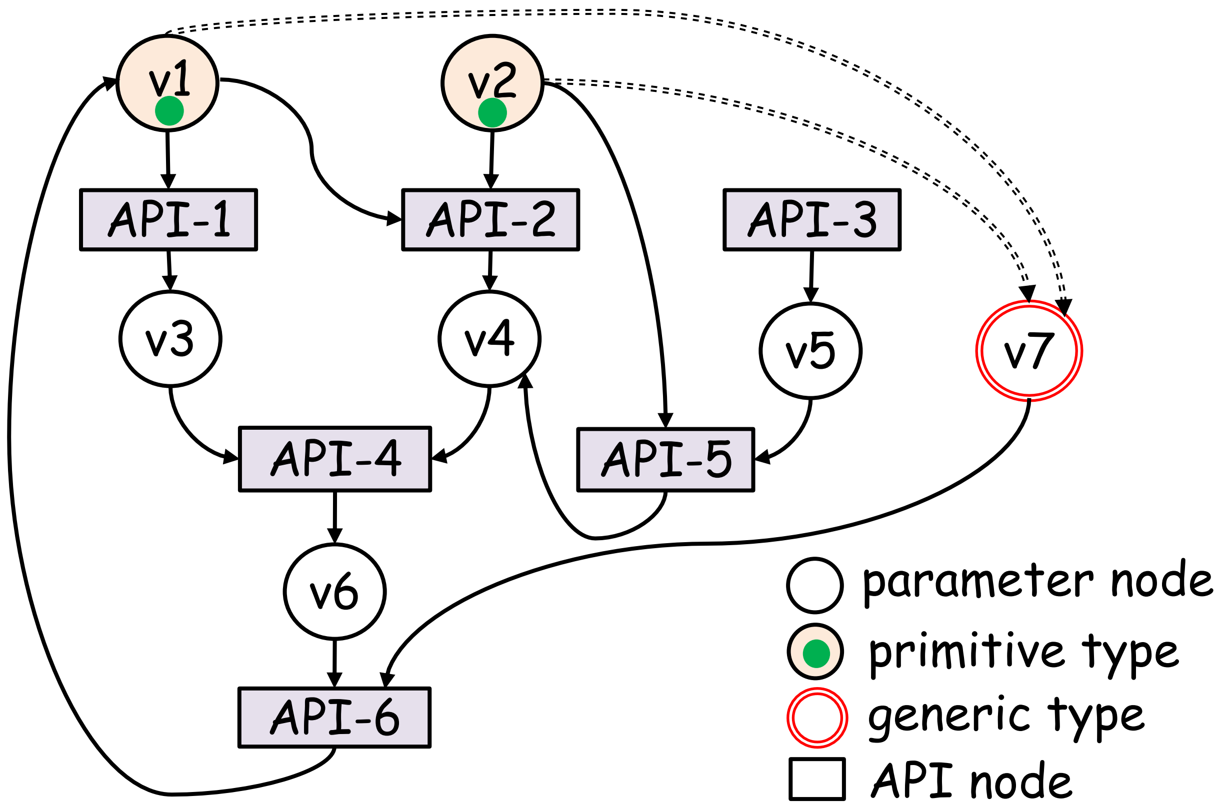
Problem

- How to fuzz Rust libraries automatically?
- We need fuzz targets for library APIs
- How to generate fuzz target automatically?

```rust
#[macro_use]
extern crate afl;
fn genvec(s:&mut String)->Vec<u8>{ ...}
fn main(){
    fuzz!(|data: &[u8]| {
        let mut s = String::from(&data);
        let v = genvec(&mut s);
    });
}
```

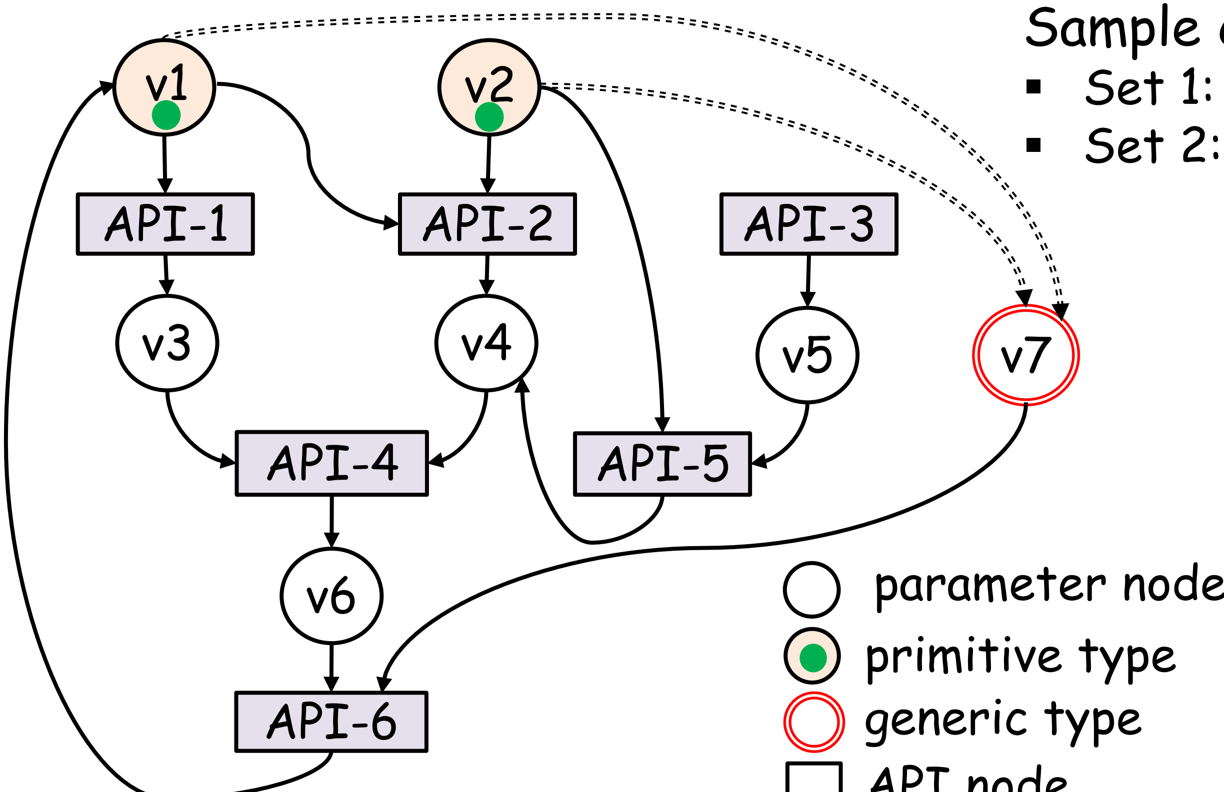
Modeling the Dependencies of APIs

- Use PetriNet



How to Traverse the Graph?

- Objective: high API coverage + efficiency
- Topological sort: generate short sequences
- Backward search: generate long sequences



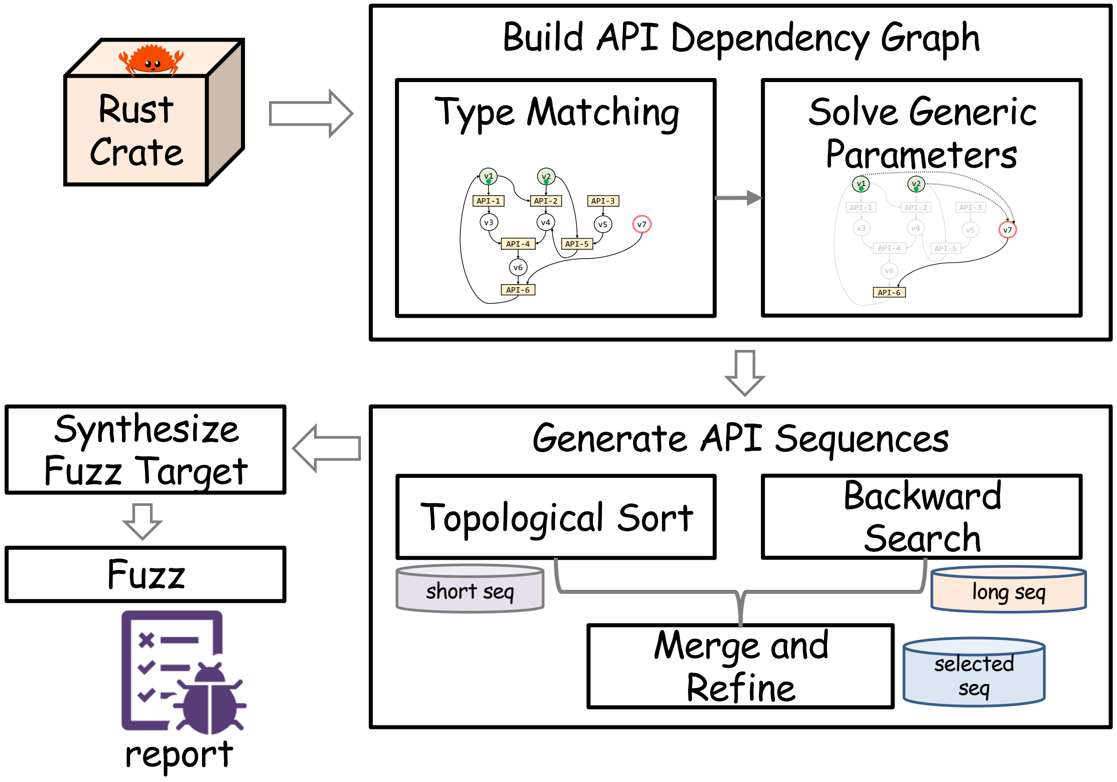
Sample choices:

- Set 1: {1-2-4-6, 3-5}
- Set 2: {1-2-4, 3-5-1-4-6}

Sequence Selection

- Selecting the sequence that contains the most uncovered API nodes
- Selecting the sequence that can activate the most uncovered parameter nodes
- Selecting the shortest sequence for those con-tribute equally using the above two rules
- Randomly picking one sequence if there are multiple candidates.

Framework Overview



4. Dynamic Symbolic Execution

Concolic (Concrete + Symbolic) Execution

Dynamic Symbolic Execution

- Stores program state concretely and symbolically
- Step:
  1) Start with random input values
  2) Keep track of both concrete values and symbolic constraints
  3) Use concrete values to simplify symbolic constraints
  4) Solve the constraints

An Illustrative Example

```
fn foo(x:usize)->usize{
    2*x
}

fn test(x:usize, y:usize){
    let z = foo(y);
    if (z == x){
        if (x > y+10){
            unreachable!();
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 22$	$x = x_0$	$2{*}y_0 \mathrel{!=} x_0$
$y = 7$	$y = y_0$	
$z = 14$	$z = 2{*}y_0$	

Solve: $2{*}y_0 == x_0$
Solution: $x_0 = 2, y_0 = 1$

An Illustrative Example

```
fn foo(x:usize)->usize{
    2*x
}

fn test(x:usize, y:usize){
    let z = foo(y);
    if (z == x){
        if (x > y+10){
            unreachable!();
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 2	$x = x_0$	$2*y_0 == x_0$
y = 1	$y = y_0$	$x_0 <= y_0+10$
z = 2	$z = 2*y_0$	

Solve: $(2*y_0 == x_0)$ and $(x_0 > y_0+10)$
Solution: $x_0 = 30$, $y_0 = 15$

An Illustrative Example

```
fn foo(x:usize)->usize{
    2*x
}

fn test(x:usize, y:usize){
    let z = foo(y);
    if (z == x){
        if (x > y+10){
            unreachable!();
        }
    }
}
```

	Concrete Execution		Symbolic Execution
concrete state		symbolic state	path condition
x = 30		$x = x_0$	$2^*y_0 == x_0$
y = 15		$y = y_0$	$x_0 > y_0+10$
z = 30		$z = 2^*y_0$	

A More Complex Example

```
fn foo(x:usize)->usize{
    hash(x)
}

fn test(x:usize, y:usize){
    let z = foo(y);
    if (z == x){
        if (x > y+10){
            unreachable!();
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	**symbolic state**	**path condition**
x = 22 y = 7	$x = x_0$ $y = y_0$	$hash(y_0)$!= x_0
z = 601…129	$z_0 = hash(y_0)$	

Solve: $hash(y_0) == x_0$
Don't know how to solve! Stuck?
Use concrete state: replace y_0 by 7.

Solve: 601…129 == x_0
Solution: x_0 = 601…129, y_0 = 7

A Third Example

```
fn foo(x:usize)->usize{
    hash(x)
}

fn test(x:usize, y:usize){
    let z = foo(y);
    if (z == foo(x)){
        if (x > y+10){
            unreachable!();
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 22$ $y = 7$	$x = x_0$ $y = y_0$	$x_0 \mathrel{!}= y_0$ $hash(x_0) \mathrel{!}= hash(y_0)$

Solve: $x_0 \mathrel{!}= y_0$ and $hash(x_0) == hash(y_0)$
Use concrete state: replace y_0 by 7.

Solve: $x_0 \mathrel{!}= 7$ and $hash(x_0) ==$ 601…129
Use concrete state: replace x_0 by 22.

Solve: $22 \mathrel{!}= 7$ and 438…861 == 601…129
Unsatisfiable!

SE Tools Recommended

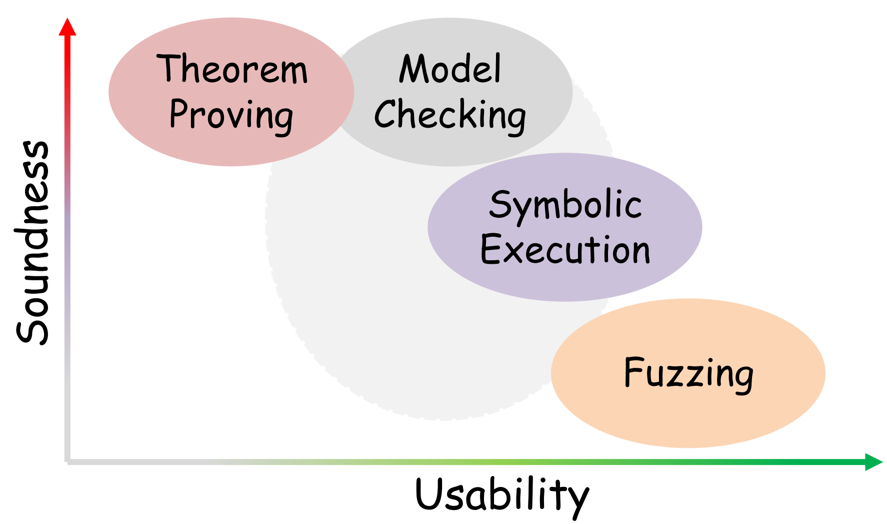
- angr: based on binary
  - https://angr.io
- KLEE: require source-code
  - https://klee.github.io

Limitation of Symbolic Execution

- Symbolic reasoning (constraint modeling/solving)
  - array/dynamic memory
  - floating-point arithmetic
  - runtime environment
  - ...
- Path-explosion (scalability): large computation tree
  - loops
  - external functions
  - ...

Comparison

- Formal methods should be sound
  - Theorem proving: prove some properties based on axioms
  - Model checking: check properties by enumerating each possible execution
    - e.g., via symbolic execution
- Fuzzing is the most easy-to-use approach



More Reference

- AFL: https://lcamtuf.coredump.cx/afl
- angr: https://angr.io
- KLEE: https://klee.github.io
- "RULF: Rust library fuzzing via API dependency graph traversal", ASE, 2021.
- "Benchmarking the capability of symbolic execution tools with logic bombs", TDSC, 2020.