

Zero Knowledge, Full Coverage: A Fuzzing Paradox

Innokentii Sennovskii

whoami



- Head of Security at Aztec Labs
- Focused on making every part of complex zero knowledge protocols secure
- Interested in fuzzing and cryptography



Agenda

- Zero Knowledge Proofs in a nutshell
- ZK Circuits
- ZK Bugs
- Fuzzing
- Fuzzing ZK Circuits

Disclaimers

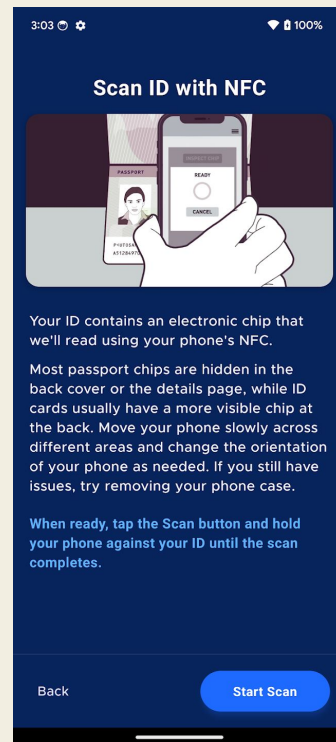
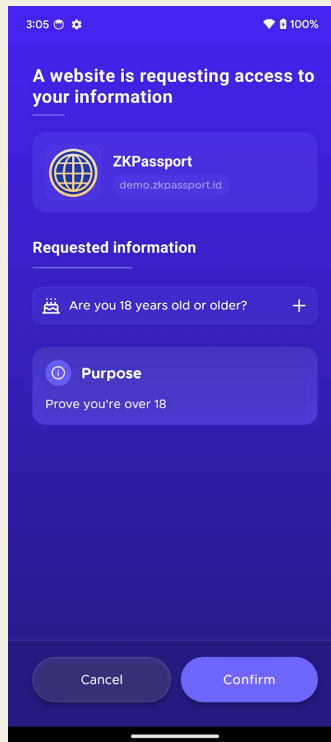
- All the views are my own
- There are some omissions and simplifications for brevity (especially in cryptographic parts)

What is a Zero-Knowledge Proof?

A protocol in which the prover can convince the verifier that some statement is true without revealing any information beyond that to the verifier

How can we use them?

- Private payments on the blockchain
- Identity checks
- zkML
- Proving email receipts
- Proving website interaction
- Proving results of binary execution



Sudoku example

The prover wants to convince the verifier that they know the solution without revealing it

	9			1			3	
		6		2		7		
			3		4			
2	1						9	8
		2	5		6	4		
	8						1	

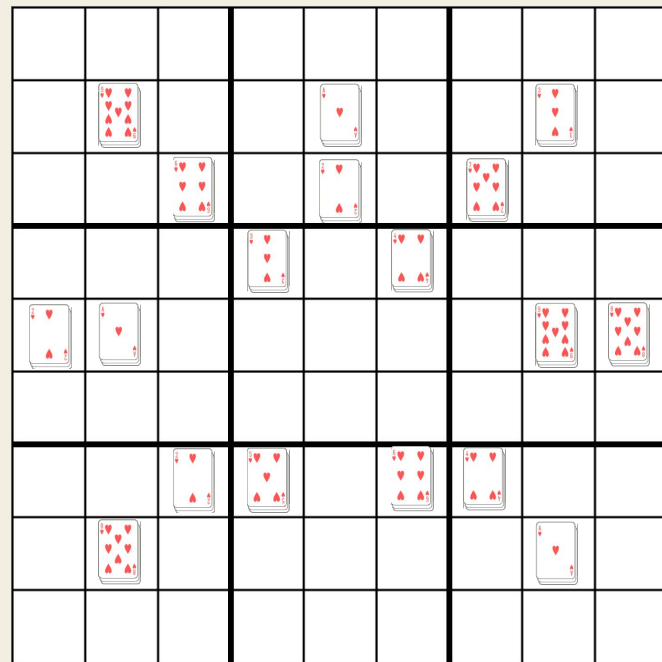
Sudoku example (Verifier step 1)

- Buys 27 card decks
- Forms 81 3-card decks (each deck has 1 card type)



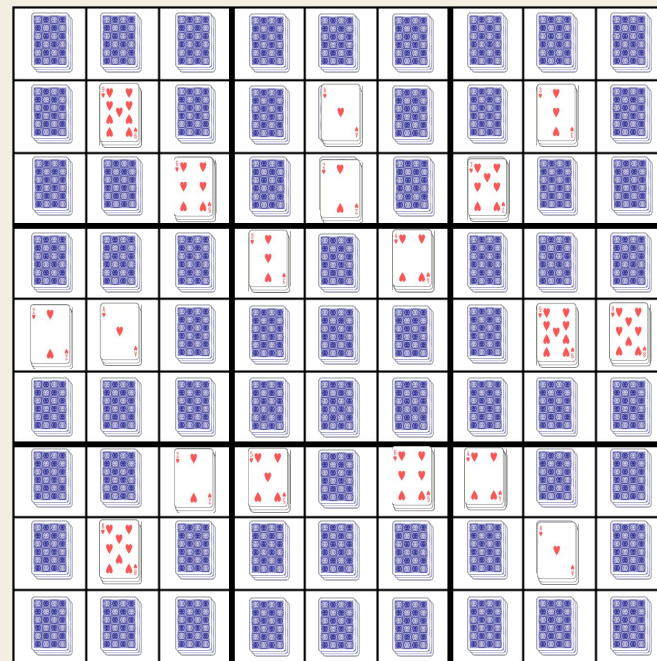
Sudoku example (Prover step)

1. Replace known numbers with decks facing up



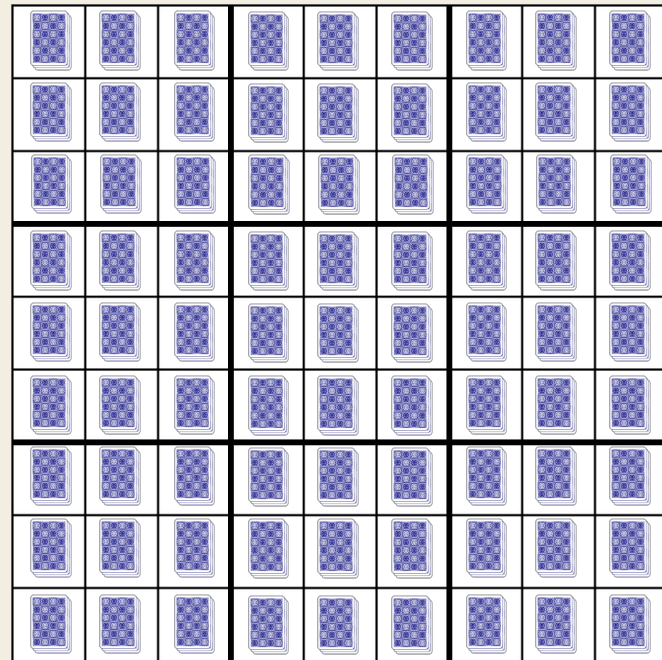
Sudoku example (Prover step)

1. Replace known numbers with decks facing up
2. Put other decks face down in accordance with the solution



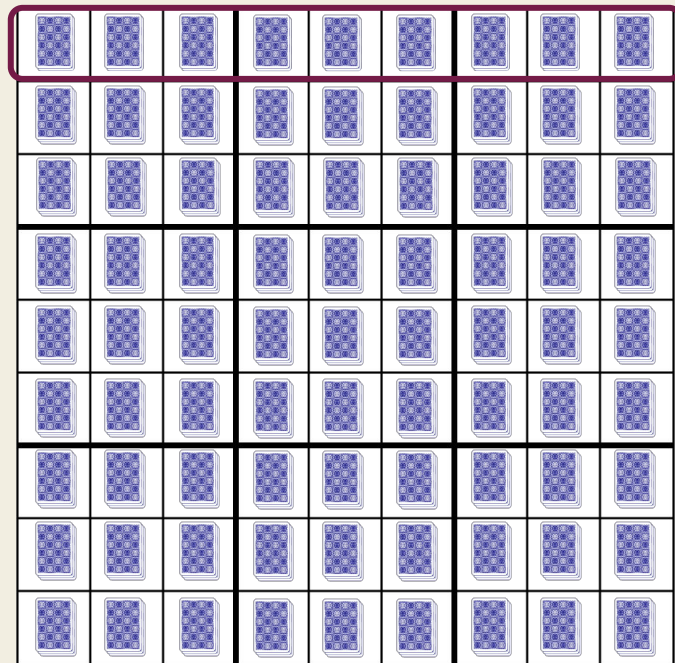
Sudoku example (Prover step)

1. Replace known numbers with decks facing up
2. Put other decks face down in accordance with the solution
3. Flip the decks facing up



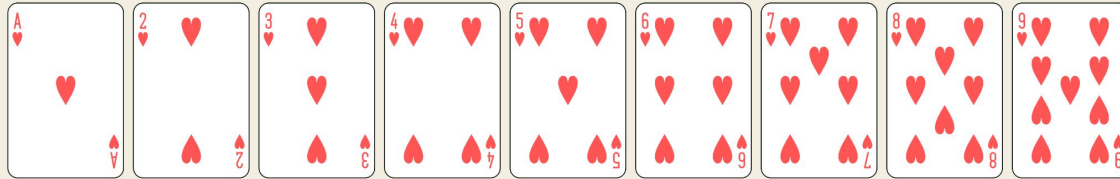
Sudoku example (Prover step)

1. Replace known numbers with decks facing up
2. Put other decks face down in accordance with the solution
3. Flip the decks facing up
4. Take 1 card from each cell in top row, shuffle, give to verifier

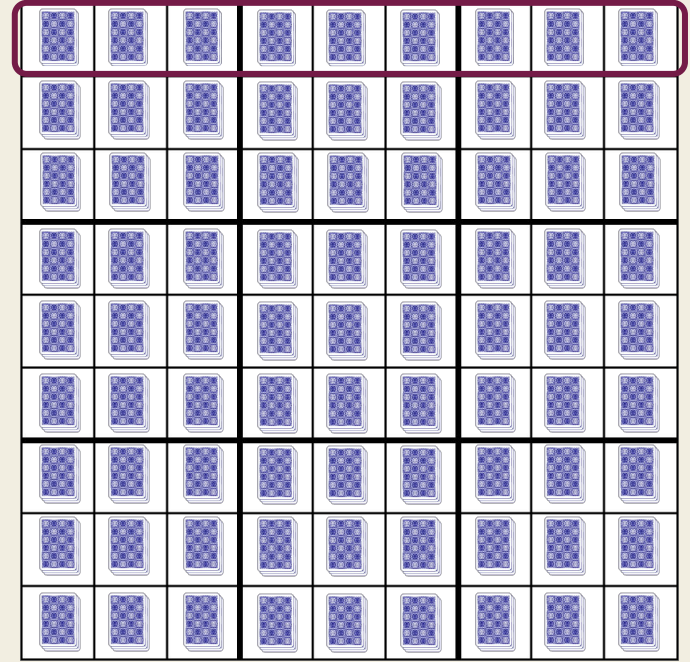
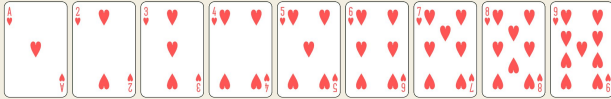


Sudoku example (Verifier step)

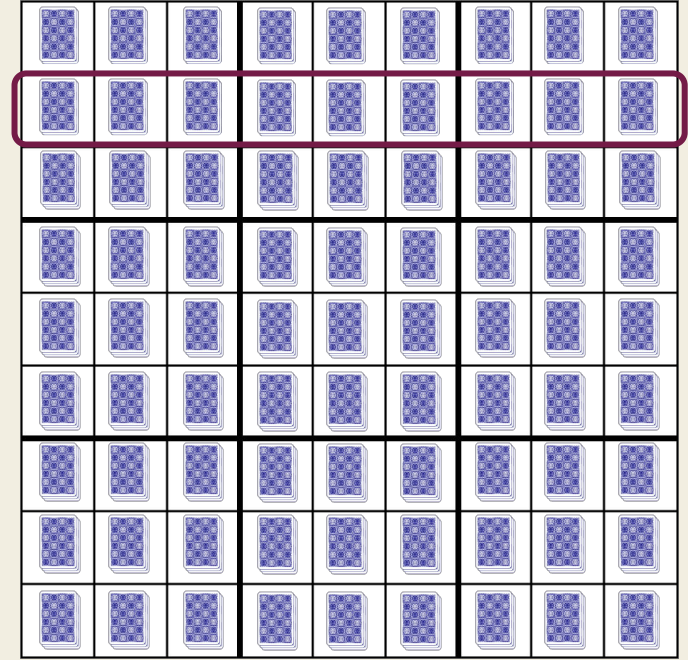
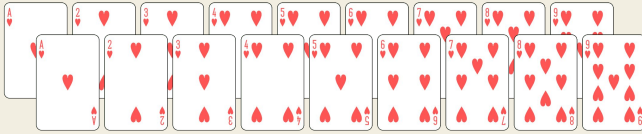
Check that the 9 received cards are:



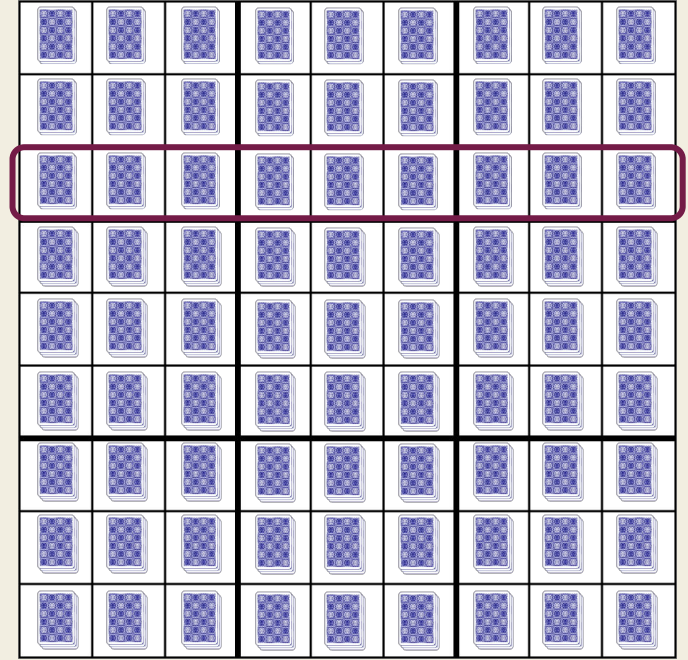
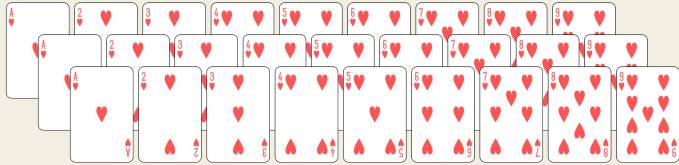
Sudoku example (...)



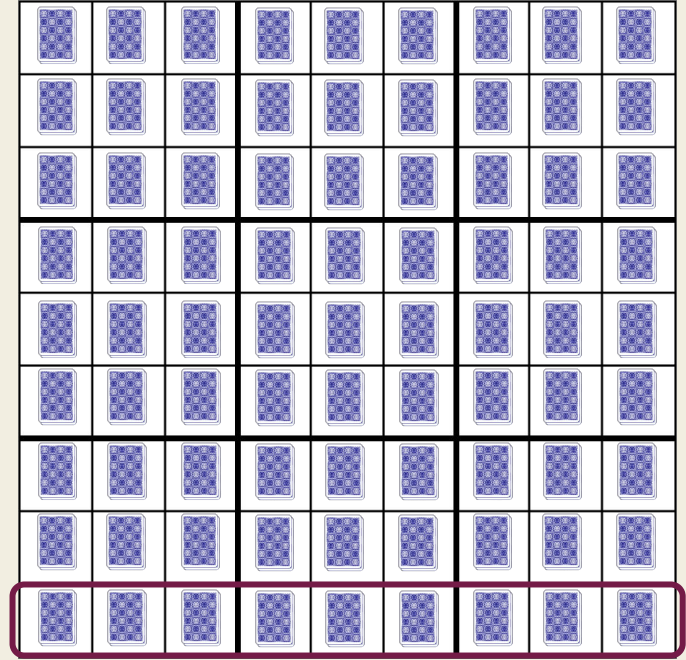
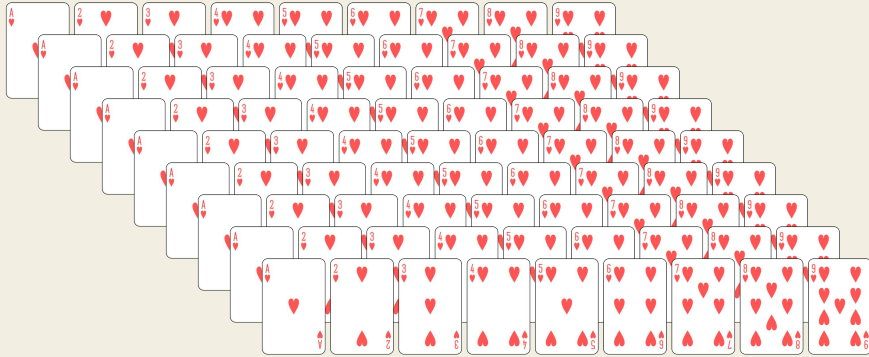
Sudoku example (...)



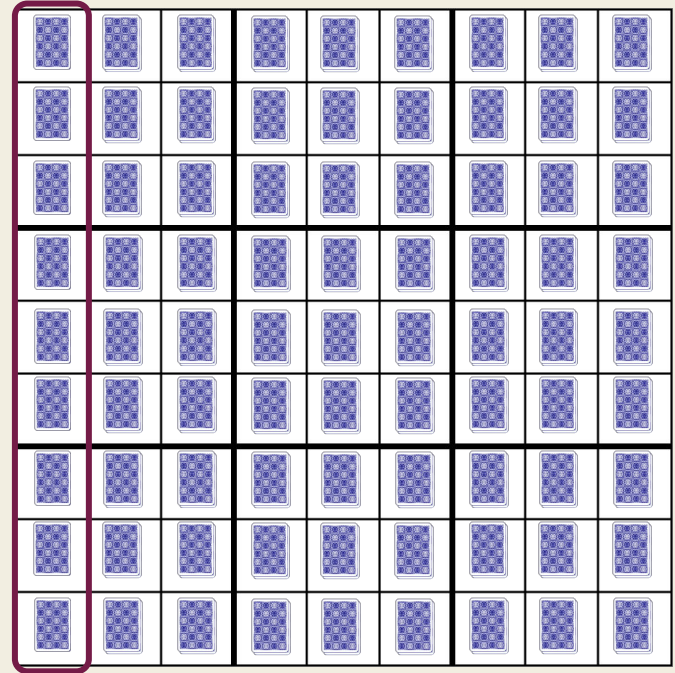
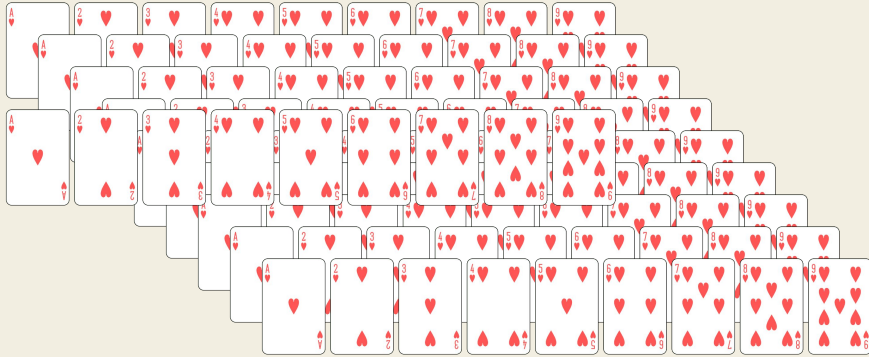
Sudoku example (...)



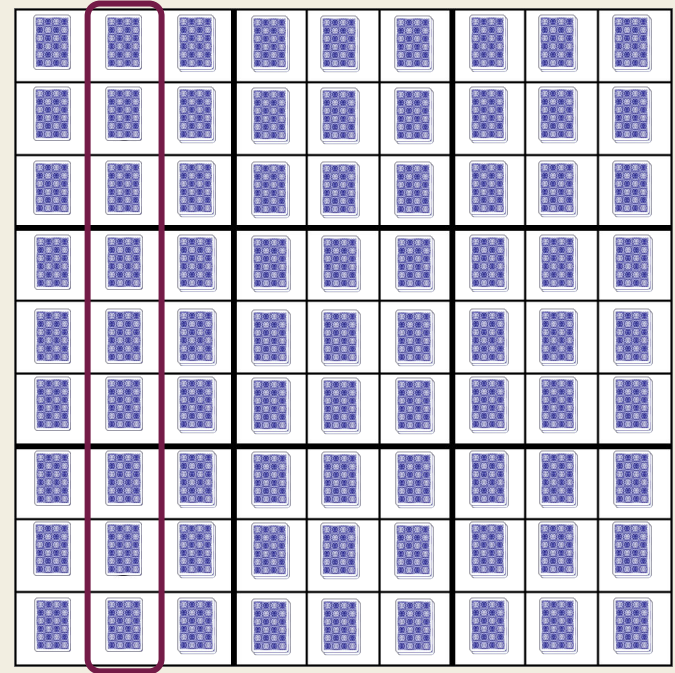
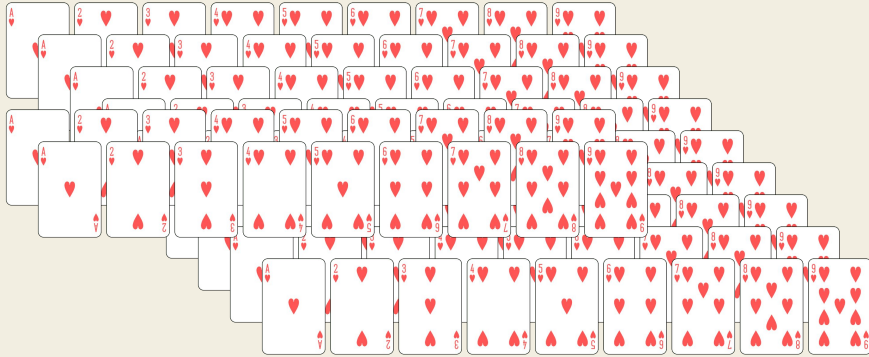
Sudoku example (...)



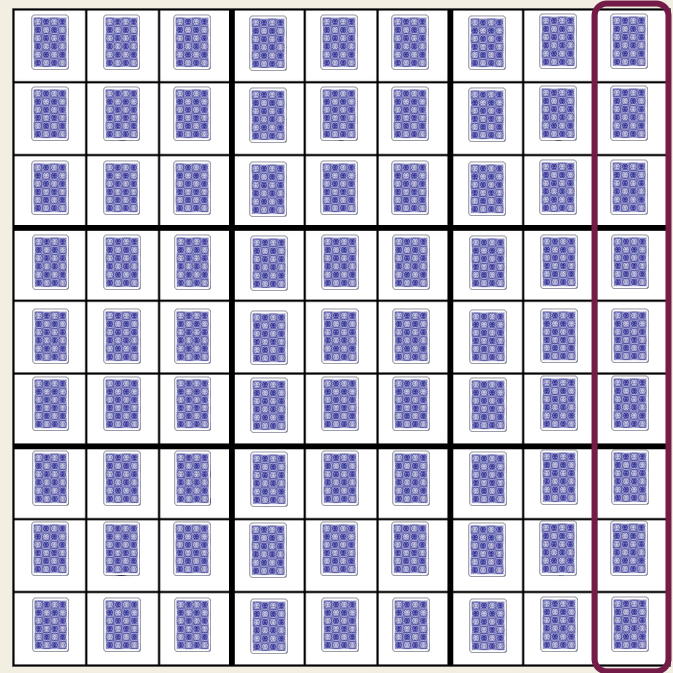
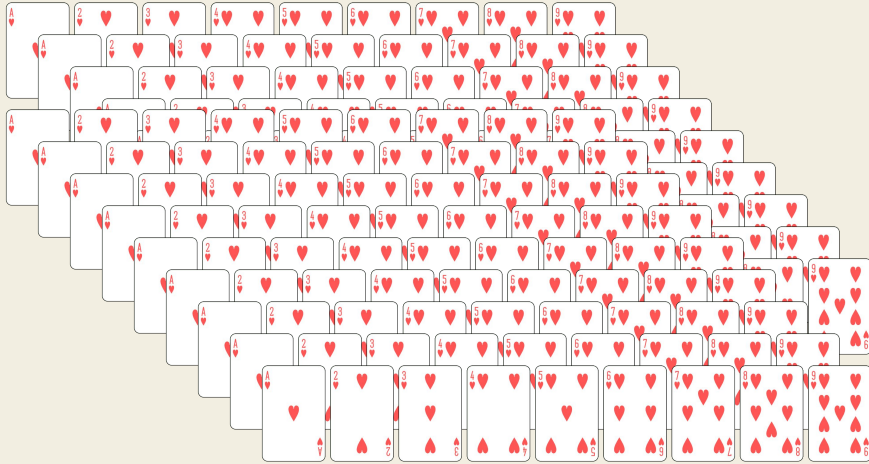
Sudoku example (...)



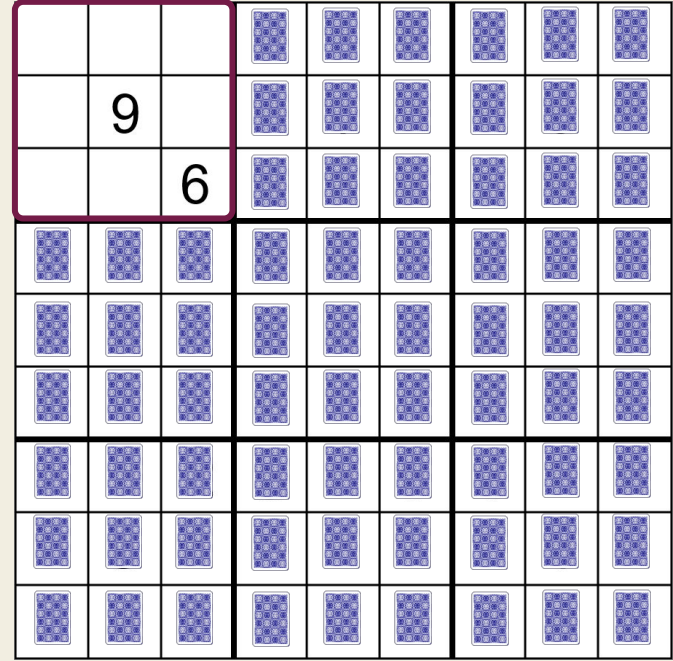
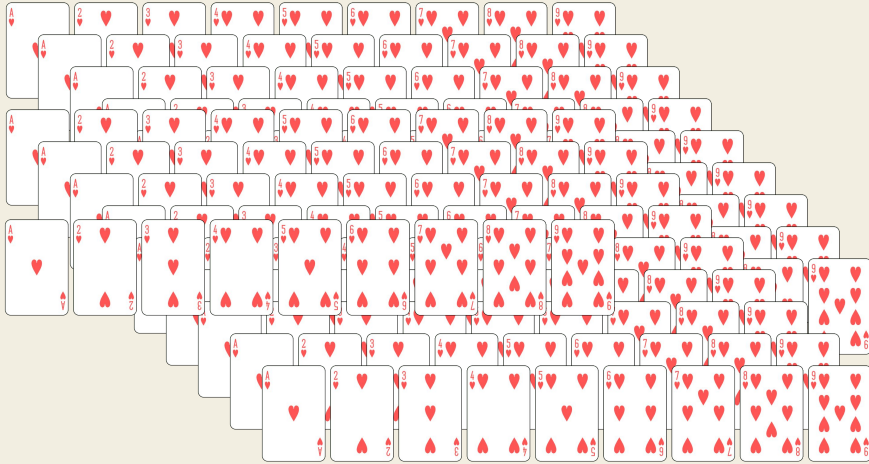
Sudoku example (...)



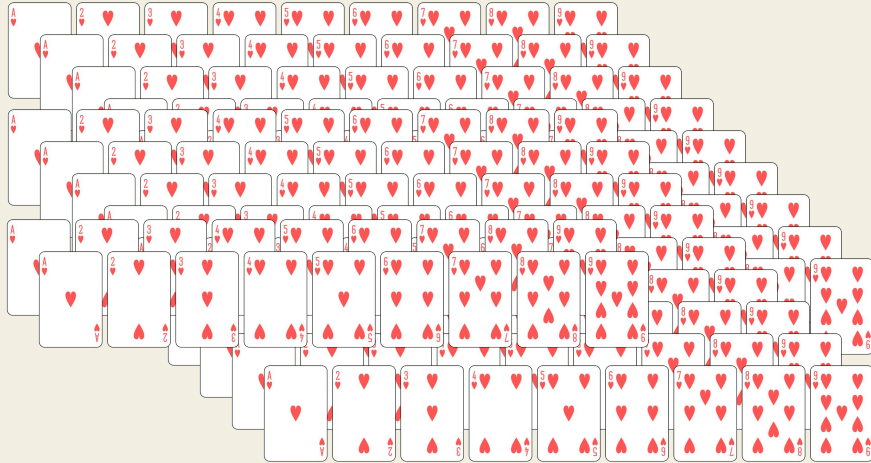
Sudoku example (...)



Sudoku example (...)

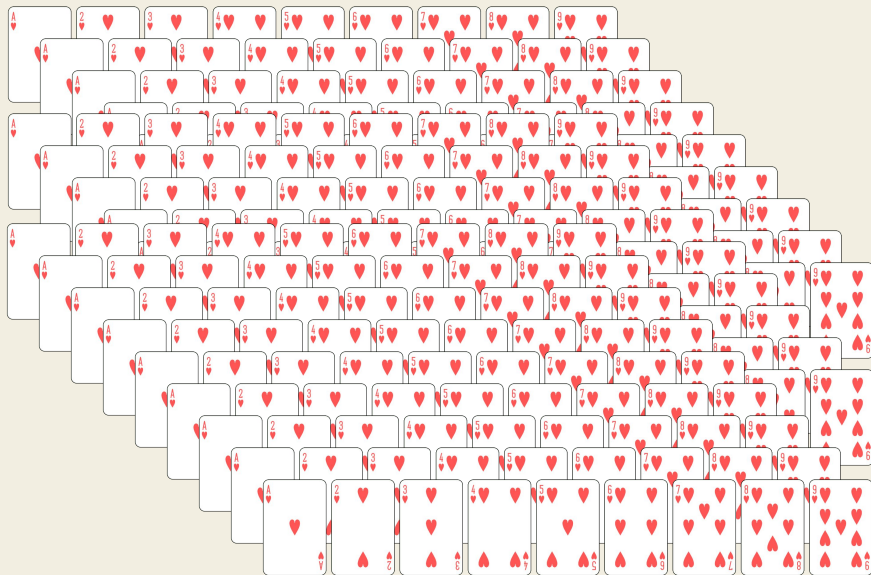


Sudoku example (...)



								
	9			1				
		6		2				
								
								
								
								
								
								

Sudoku example (...)



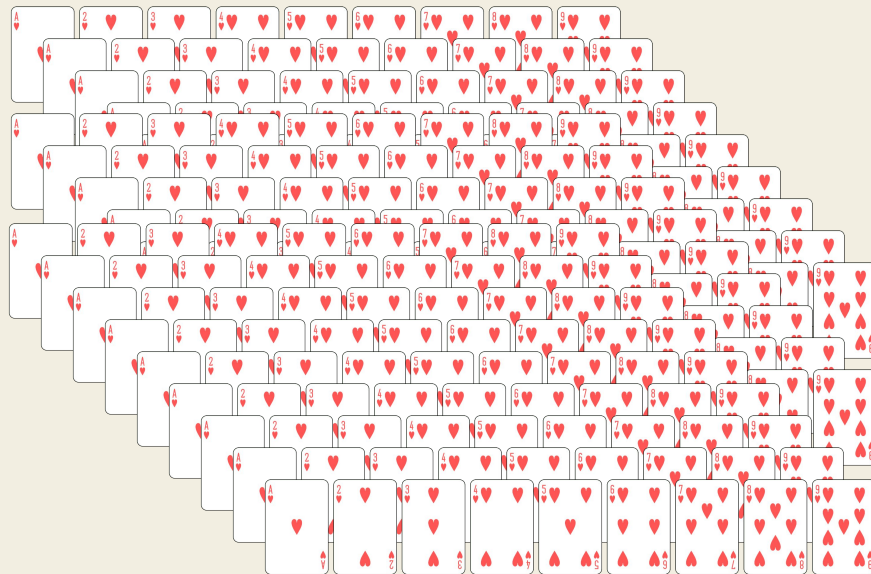
	9			1			3	
		6		2		7		
			3		4			
2	1						9	8
		2	5		6	4		
	8						1	

Sudoku example

The verifier now knows:

1. Each row, column, square had all numbers
2. The values in predefined cells were satisfied

And NOTHING beyond that



Discrete logarithm

Large secure prime numbers have a cool feature:

- Computing $y = a^x \bmod p$ is easy
- Computing $x = \log_a y \bmod p$ is “hard” (not possible with current resources)

Discrete logarithm example

Knows x

Knows $y = g^x \bmod p$, doesn't know x

Prover

Verifier

Generates random k ,
computes $r = g^k \bmod p$

r

e

Generates random
challenge e

Computes $s = k + x \cdot e$

s

Checks $g^s = r \cdot y^e =$
 $= g^k \cdot (g^x)^e = g^{k + xe} \bmod p$

Why is this Zero Knowledge?

Doesn't know x , but can predict e

Knows $y = g^x \bmod p$, doesn't know x

Prover

Verifier

Generates random k ,
computes $r = g^k \cdot y^e \bmod p$

r

e

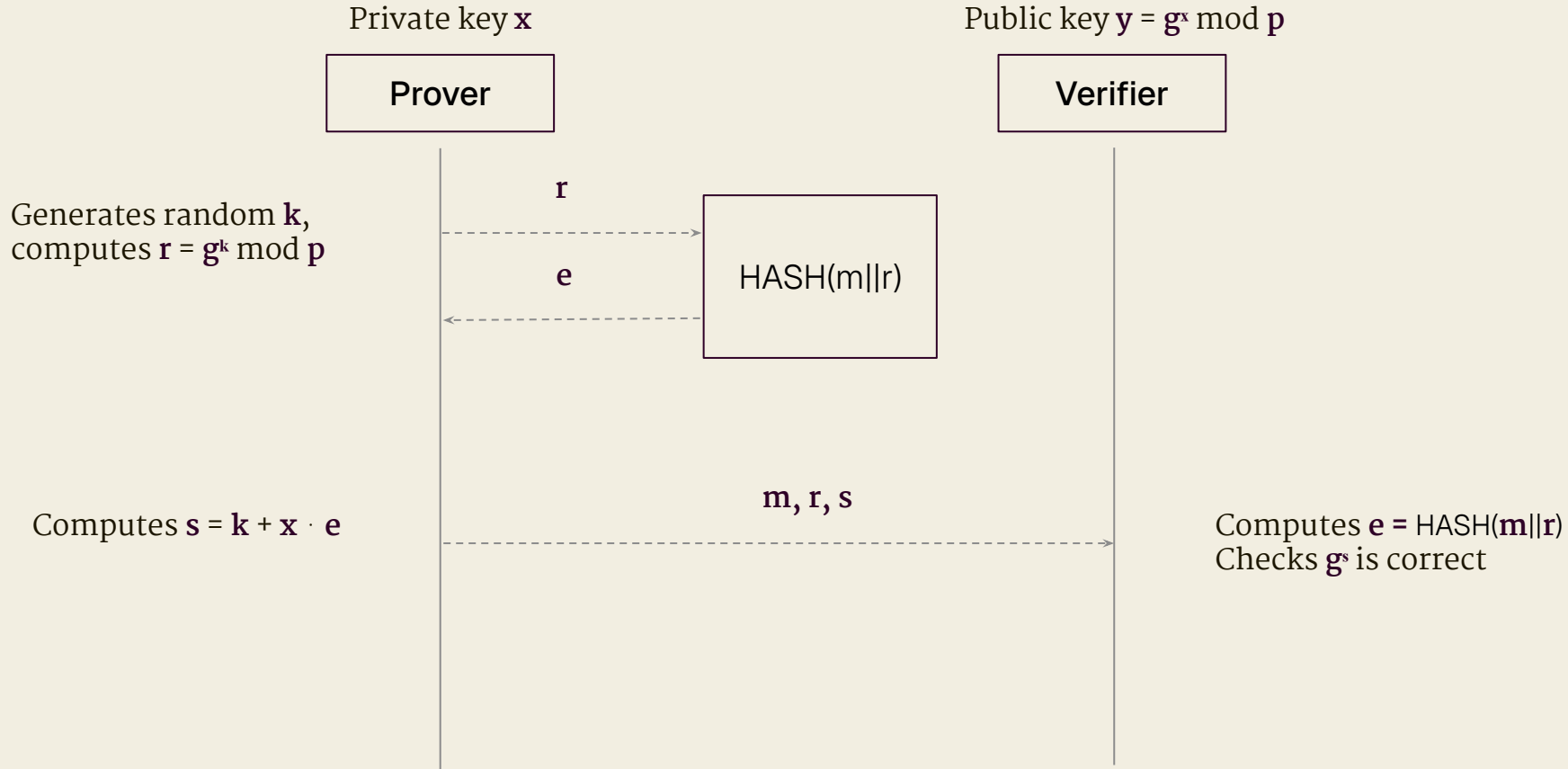
s

$s = k$

Generates predictable
challenge e

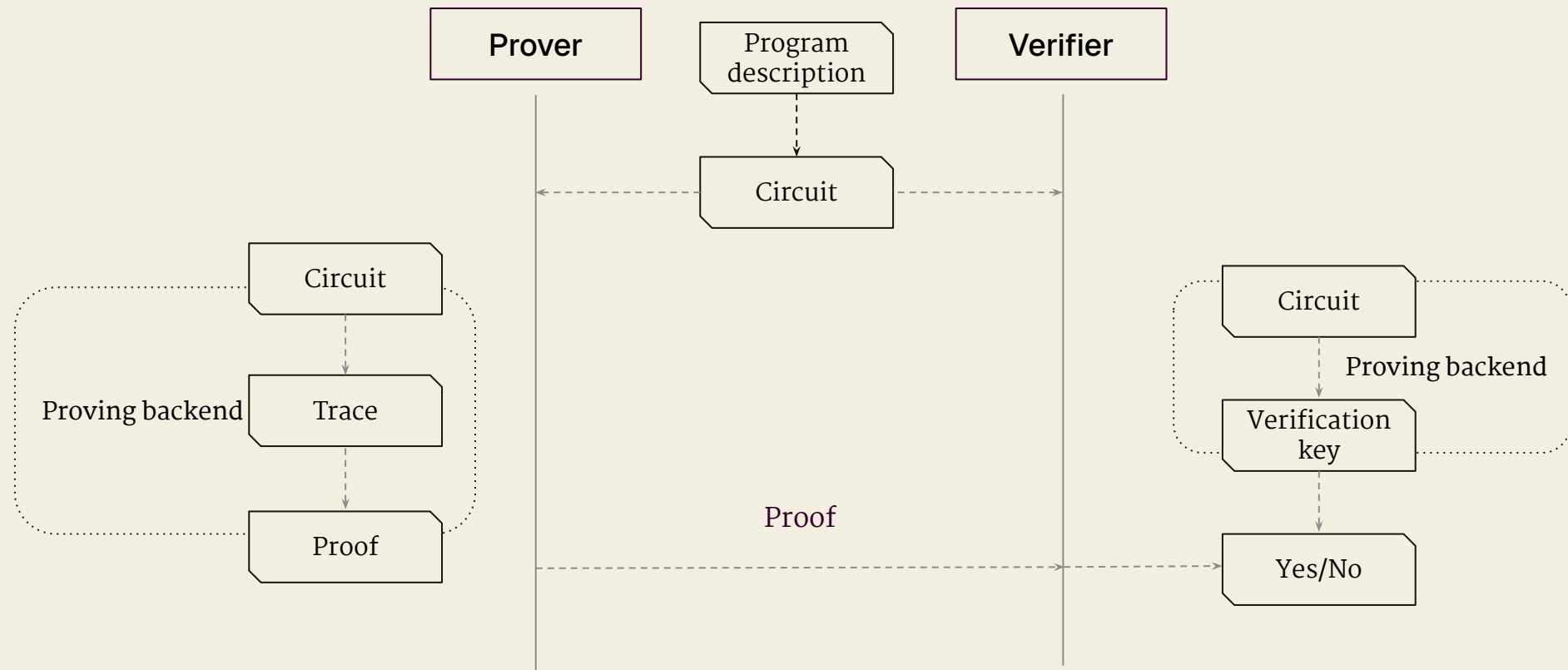
Checks $g^s = r \cdot y^e =$
 $= g^k \cdot y^e \cdot (g^x)^e = g^k \bmod p$

Schnorr's signature (noninteractive)



ZK Circuits

Proofs for complex logic



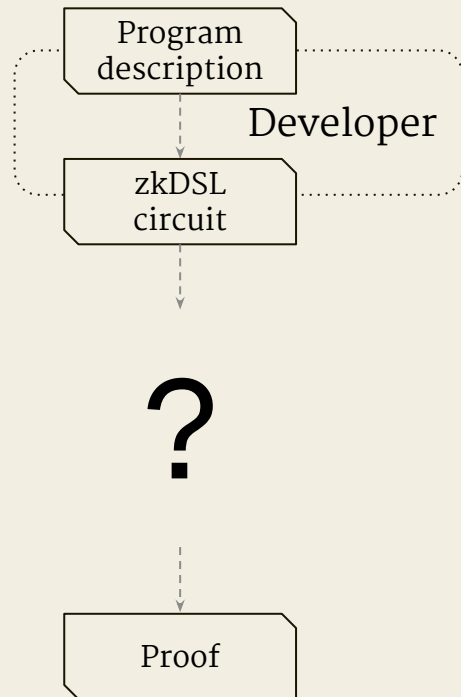
Proofs for complex logic (Rust)

```
fn fibonacci(a: u64, b: u64, steps: usize) -> u64 {  
    let mut a = a;  
    let mut b = b;  
    if steps == 0 {  
        return a;  
    }  
    if steps == 1 {  
        return b;  
    }  
    for i in 2..steps {  
        let next = a + b;  
        a = b;  
        b = next;  
    }  
    b  
}
```



Proofs for complex logic (Noir)

```
global MAX_STEPS: u32 = 3; // Just for this example
fn main(a: pub Field, b: Field, target_step: pub u32) -> pub Field {
  let mut result = a;
  let mut a = a;
  let mut b = b;
  if target_step == 1 {
    result = b;
  }
  for i in 2..MAX_STEPS + 1 {
    let temp = a;
    a = b;
    b = temp + a;
    if i == target_step {
      result = b;
    }
  }
  result
}
```



zkDSL Circuit



```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) -> pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

Visibility

- Private inputs
 - Need to stay secret
- Public inputs
 - Sent to the verifier

Outputs are always public



zkDSL Circuit (Noir)



```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) -> pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

Basic type

Instead of 32/64 bit logic, circuits operate in a field of integers modulo prime number, for example

```
0x30644e72e131a029b85045b68181585d2833e
84879b9709143e1f593f0000001
```

Everything else is defined on top of them



zkDSL Circuit



```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) -> pub Field {
  let mut result = a;
  let mut a = a;
  let mut b = b;
  if target_step == 1 {
    result = b;
  }
  for i in 2..MAX_STEPS + 1 {
    let temp = a;
    a = b;
    b = temp + a;
    if i == target_step {
      result = b;
    }
  }
  result
}
```

Finite execution length

A circuit always has the same size, regardless of inputs.

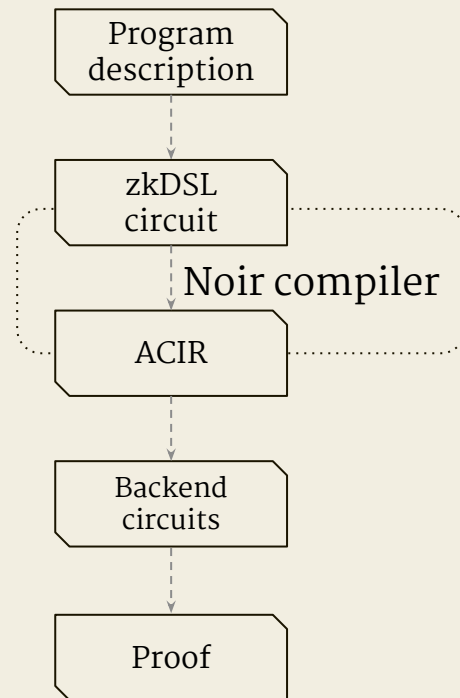
Loops are fixed-length



Algebraic Circuit Intermediate Representation



```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs: [w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs: [w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```



ACIR Expressions (ASSERT)

Enforces an algebraic relation between variables

Like sudoku, the prover has to provide the values that satisfy all equations at once

Most of the time the prover can compute a new variable from the previous ones

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

ACIR Expressions (BLACKBOX::RANGE)



Blackbox functions are provided by the proving backend

Range enforces a rule “variable is between 0 and $2^{\text{bits}} - 1$ ”

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```



ACIR Expressions (Brillig Call)

Efficiently compute a new value in an unconstrained VM

For example, we want to show that $b = a^2$

Computing a in the circuit is costly (lots of ASSERTs)

Computing it outside and then just showing the relationship is cheap (1 ASSERT)

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 {
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 { // i == 2
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 { // i == 2
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 { // i == 2
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 { // i == 3
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Noir <-> ACIR

```
global MAX_STEPS: u32 = 3;
fn main(a: pub Field, b: Field, target_step: pub u32) ->
pub Field {
    let mut result = a;
    let mut a = a;
    let mut b = b;
    if target_step == 1 {
        result = b;
    }
    for i in 2..MAX_STEPS + 1 { // i == 3
        let temp = a;
        a = b;
        b = temp + a;
        if i == target_step {
            result = b;
        }
    }
    result
}
```

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

ZK Bugs

Constraints

ASSERTS and Blackbox functions that enforce rules on the variables

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 + 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Witness

A set of variable values satisfying the constraints

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs:
[w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs:
[w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```

Important ZK Circuit Property

The prover can use any witness satisfying the constraints to create a valid proof that will be accepted by the verifier



Soundness bugs

Prover can convince the verifier that a statement is true, when it is not

Wrong sudoku solution is accepted

Examples:

- Necessary constraints are missing
- Logic issues
- Improper use of unconstrained functions



Soundness bug example

```
struct u128 {
    data: [u64; 2],
}

impl Eq for u128 {
    fn eq(self, other: Self) -> bool {
        (self.data[0] as Field - other.data[0] as Field)
            * (self.data[1] as Field - other.data[1] as
Field)
            == 0
    }
}

fn main(x: u128, y: pub u128) {
    assert(x == y);
}

// [0,0] == [0,1]
```

```
func 0
private parameters: [w0, w1]
public parameters: [w2, w3]
return values: []
BLACKBOX::RANGE input: w0, bits: 64
BLACKBOX::RANGE input: w1, bits: 64
BLACKBOX::RANGE input: w2, bits: 64
BLACKBOX::RANGE input: w3, bits: 64
ASSERT w4 = w0 - w2
ASSERT w5 = w1 - w3
ASSERT 0 = w4*w5
```

Completeness bugs

Prover can't convince the verifier that a statement is true, when it is

Correct sudoku solution is not accepted

Examples:

- Forgotten edgecases
- Developer incorrectly transfers regular program logic to field logic



Completeness bug example

```
fn main(x: u8, y: pub u8) {  
    assert_eq(x + y, 0)  
}
```

```
// The only solution is 0,0 because  
// the developer didn't use wrapping_add
```

```
func 0  
private parameters: [w0]  
public parameters: [w1]  
return values: []  
BLACKBOX::RANGE input: w0, bits: 8  
BLACKBOX::RANGE input: w1, bits: 8  
ASSERT w2 = w0 + w1  
ASSERT w2 = 0
```

Zero-knowledge bugs

Prover reveals something extra about the solution

Values of some sudoku cells are revealed

Examples:

- Pub modifier is assigned to wrong inputs
- Something is leaked in the proving backend



Traditional ZK circuit bug detection

Soundness

- Unit testing
- E2E testing
- Formal verification
- Audits

Completeness

- Unit testing
- E2E testing
- Formal verification
- Audits

Leakage

- Static analysis (rarely)
- Audits

Problems

Unit testing

- Depends too much on the developer
- Doesn't test component interaction
- Hard to test all the edgecases
- Doesn't test unknown unknowns

E2E testing

- Follows predetermined paths
- Hard to test all the edgecases
- Doesn't test unknown unknowns

Formal Verification

- Usually requires a lot of human effort
- Stalls on larger circuits

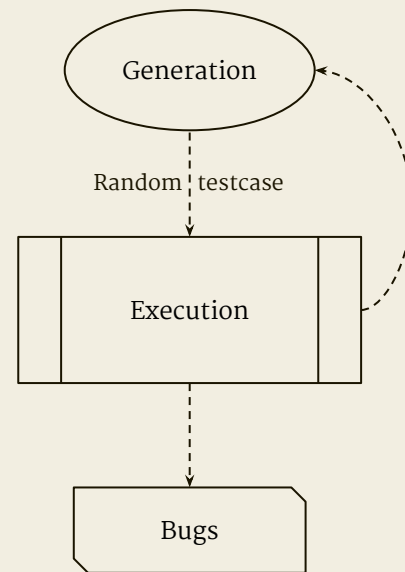
Audits

- Very human-dependent
- Not scalable
- Cost a ton

Fuzzing

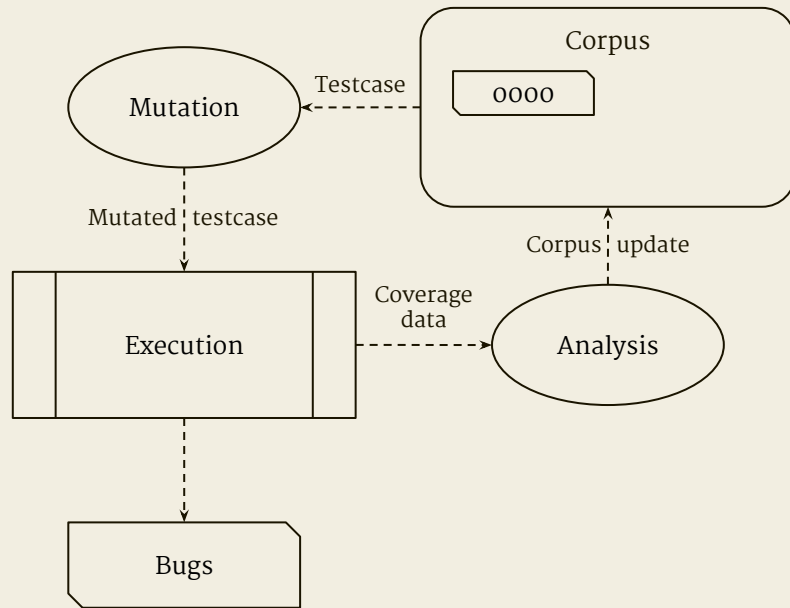
Blackbox fuzzing

```
fn fuzzing_harness(input: [u8; 4]) {  
    if input[0] == b'a' { // ~(1/256)  
        if input[1] == b'b' { // ~(1/65536)  
            if input[2] == b'c' { // ~(1/16777216)  
                if input[3] == b'd' { // ~(1/4294967296)  
                    panic!("Triggered!");  
                }  
            }  
        }  
    }  
}
```



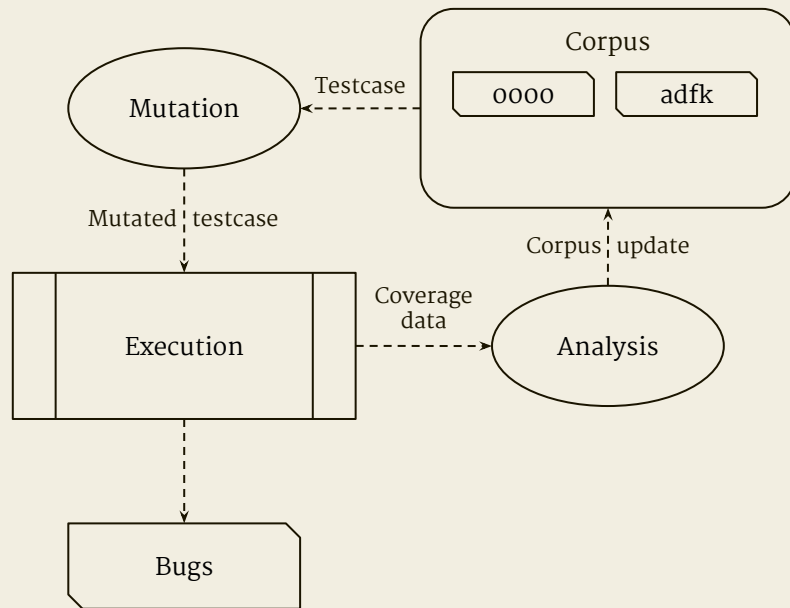
Greybox fuzzing

```
fn fuzzing_harness(input: [u8; 4]) {  
    if input[0] == b'a' { // ~(1/256)  
        if input[1] == b'b' { // ~(1/65536)  
            if input[2] == b'c' { // ~(1/16777216)  
                if input[3] == b'd' { // ~(1/4294967296)  
                    panic!("Triggered!");  
                }  
            }  
        }  
    }  
}
```



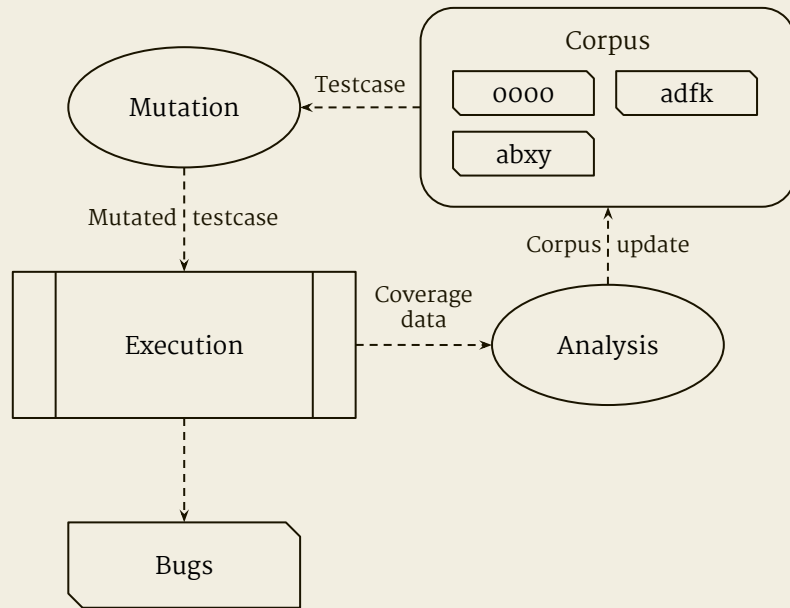
Greybox fuzzing

```
fn fuzzing_harness(input: [u8; 4]) {  
    if input[0] == b'a' { // Found!  
        if input[1] == b'b' { // ~(1/256)  
            if input[2] == b'c' { // ~(1/65536)  
                if input[3] == b'd' { // ~(1/16777216)  
                    panic!("Triggered!");  
                }  
            }  
        }  
    }  
}
```



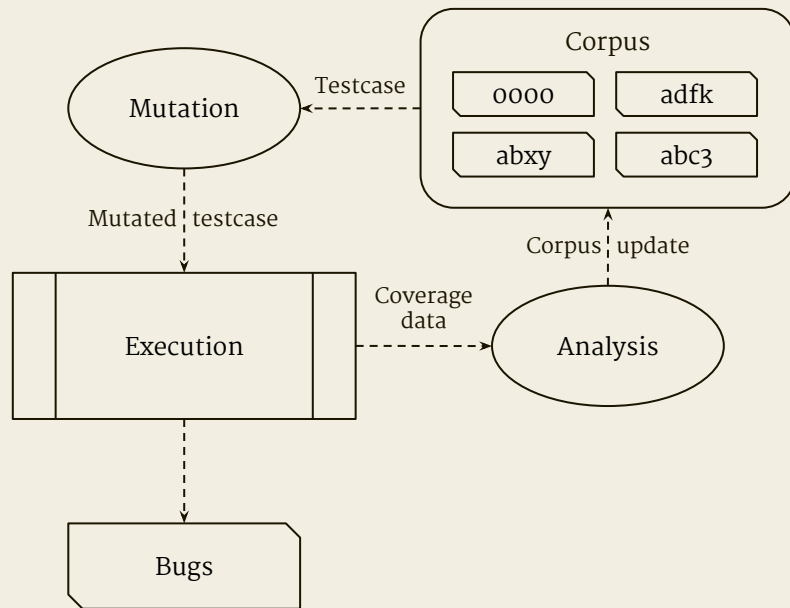
Greybox fuzzing

```
fn fuzzing_harness(input: [u8; 4]) {  
    if input[0] == b'a' { // Found!  
        if input[1] == b'b' { // Found!  
            if input[2] == b'c' { // ~(1/256)  
                if input[3] == b'd' { // ~(1/65536)  
                    panic!("Triggered!");  
                }  
            }  
        }  
    }  
}
```



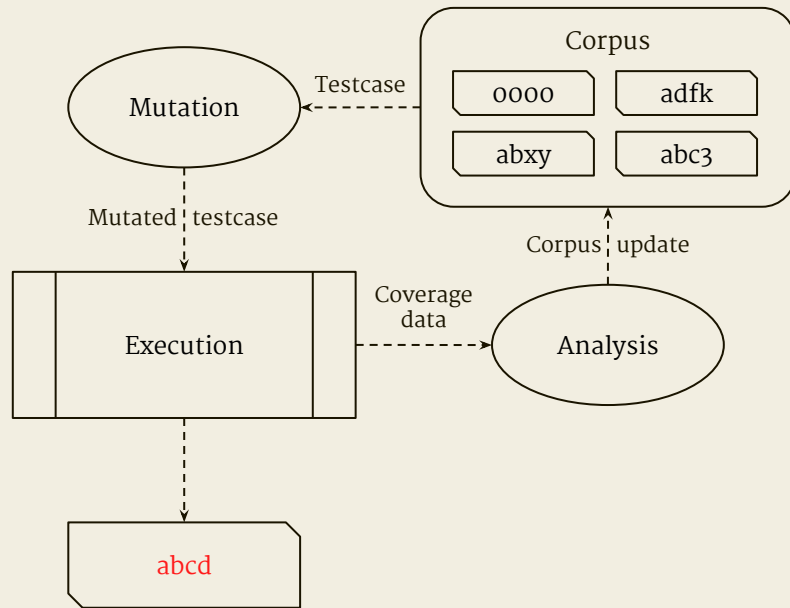
Greybox fuzzing

```
fn fuzzing_harness(input: [u8; 4]) {  
    if input[0] == b'a' { // Found!  
        if input[1] == b'b' { // Found!  
            if input[2] == b'c' { // Found!  
                if input[3] == b'd' { // ~(1/256)  
                    panic!("Triggered!");  
                }  
            }  
        }  
    }  
}
```



Greybox fuzzing

```
fn fuzzing_harness(input: [u8; 4]) {  
    if input[0] == b'a' { // Found!  
        if input[1] == b'b' { // Found!  
            if input[2] == b'c' { // Found!  
                if input[3] == b'd' { // Found!  
                    panic!("Triggered!");  
                }  
            }  
        }  
    }  
}
```



But circuits don't have branching!

```
func 0
private parameters: [w1]
public parameters: [w0, w2]
return values: [w3]
BLACKBOX::RANGE input: w2, bits: 32
BRILLIG CALL func: 0, inputs: [w2 - 1], outputs: [w4]
ASSERT w5 = -w2*w4 + w4 + 1
ASSERT w5 = w2*w5
BRILLIG CALL func: 0, inputs: [-w2 + 2], outputs: [w6]
ASSERT w7 = w2*w6 - 2*w6 + 1
ASSERT 0 = -w2*w7 + 2*w7
ASSERT w8 = -w7 + 1
ASSERT w9 = -w0*w5 + w1*w5 + w0
BRILLIG CALL func: 0, inputs: [-w2 + 3], outputs: [w10]
ASSERT w11 = w2*w10 - 3*w10 + 1
ASSERT 0 = -w2*w11 + 3*w11
ASSERT w12 = -w11 + 1
ASSERT w13 = w0*w7 + w1*w7 + w8*w9
ASSERT w3 = w0*w11 + 2*w1*w11 + w12*w13
```



Fuzzing ZK Circuits

Noir supports unconstrained functions

```
unconstrained fn sum_bits(a: u64) -> u8 {  
    let mut bit_sum: u8 = 0;  
    let mut a = a;  
    while a != 0 {  
        if a & 1 == 1 {  
            bit_sum += 1;  
        }  
        a >>= 1;  
    }  
    bit_sum  
}  
fn main(x: u64) -> pub u8 {  
    unsafe {  
        sum_bits(x)  
    }  
}
```

```
unconstrained func 0: sum_bits  
0: @2 = const u32 1  
1: @1 = const u32 32838  
2: @0 = const u32 70  
3: sp[2] = const u32 1  
4: sp[3] = const u32 0  
5: @68 = calldata copy [sp[3]; sp[2]]  
6: @68 = cast @68 to u64  
7: sp[1] = @68  
8: call 14  
9: call 15  
10: @69 = sp[1]  
11: sp[2] = const u32 69  
12: sp[3] = const u32 1  
13: stop &[sp[2]; sp[3]]  
14: return  
15: call 48  
16: sp[2] = @1  
17: @1 = u32 add @1, @2  
18: sp[3] = const u8 0  
19: store sp[3] at sp[2]  
20: sp[3] = @1  
21: @1 = u32 add @1, @2  
22: store sp[1] at sp[3]  
23: sp[1] = const u64 0  
24: sp[4] = const u64 1  
25: sp[5] = const u8 1  
26: jump to 27  
27: sp[6] = load sp[3]  
28: sp[7] = u64 eq sp[6], sp[1]  
29: jump if sp[7] to 46  
30: jump to 31
```

...



Brillig VM

- Field, u8, u16, u32, u64, u128 types
- Contiguous memory
- **Branching!!!**

```
unconstrained func 0: sum_bits
0: @2 = const u32 1
1: @1 = const u32 32838
2: @0 = const u32 70
3: sp[2] = const u32 1
4: sp[3] = const u32 0
5: @68 = calldata copy [sp[3]; sp[2]]
6: @68 = cast @68 to u64
7: sp[1] = @68
8: call 14
9: call 15
10: @69 = sp[1]
11: sp[2] = const u32 69
12: sp[3] = const u32 1
13: stop &[sp[2]; sp[3]]
14: return
15: call 48
16: sp[2] = @1
17: @1 = u32 add @1, @2
18: sp[3] = const u8 0
19: store sp[3] at sp[2]
20: sp[3] = @1
21: @1 = u32 add @1, @2
22: store sp[1] at sp[3]
23: sp[1] = const u64 0
24: sp[4] = const u64 1
25: sp[5] = const u8 1
26: jump to 27
27: sp[6] = load sp[3]
28: sp[7] = u64 eq sp[6], sp[1]
29: jump if sp[7] to 46
30: jump to 31
```

...

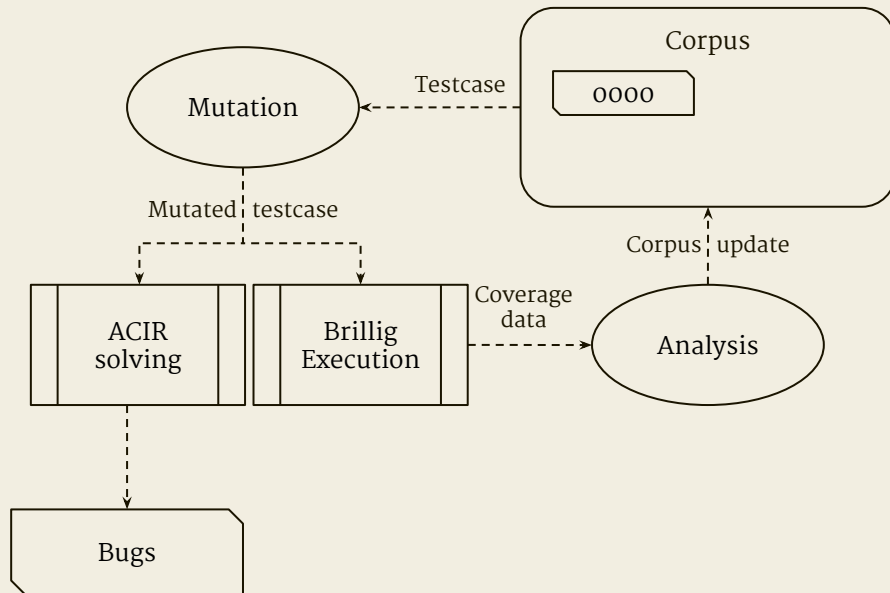


Solution

1. Compile fuzzing harness as:
 - a. ACIR
 - b. Brillig
2. On each loop:
 - a. Solve witness for ACIR, detect bugs
 - b. Run brillig VM, extract coverage

Coverage types:

1. Branching (jmp if)
2. Value proximity (eq, lt)



Fuzzing harness

```
#[fuzz]
fn ladder(a: [u8; 10]) {
    if a[0] == 1 {
        if a[1] == 2 {
            if a[2] == 3 {
                if a[3] == 4 {
                    if a[4] == 5 {
                        if a[5] == 6 {
                            if a[6] == 7 {
                                if a[7] == 8 {
                                    if a[8] == 9 {
                                        if a[9] == 10 {
                                            assert(false);
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Fuzzing process



Computation complexity

```
fn main(x: u64, y: pub u64) -> pub u64 {  
    x >> y  
}
```

```
func 0  
private parameters: [w0]  
public parameters: [w1]  
return values: [w2]  
BLACKBOX::RANGE input: w0, bits: 64  
BLACKBOX::RANGE input: w1, bits: 64  
BRILLIG CALL func: 0, inputs: [w1 + 18446744073709551552,  
18446744073709551616], outputs: [w3, w4]  
BLACKBOX::RANGE input: w4, bits: 64  
ASSERT w4 = w1 - 18446744073709551616*w3 + 18446744073709551552  
ASSERT w3 = 0  
BRILLIG CALL func: 1, inputs: [w1, 6, 2], outputs: [[w5, w6, w7,  
w8, w9, w10]]  
BLACKBOX::RANGE input: w5, bits: 1  
BLACKBOX::RANGE input: w6, bits: 1  
BLACKBOX::RANGE input: w7, bits: 1  
BLACKBOX::RANGE input: w8, bits: 1  
BLACKBOX::RANGE input: w9, bits: 1  
BLACKBOX::RANGE input: w10, bits: 1  
ASSERT w5 = w1 - 2*w6 - 4*w7 - 8*w8 - 16*w9 - 32*w10  
ASSERT w11 = w10*w10 + 2*w10 + 1  
ASSERT w12 = -w9 + 1  
ASSERT w13 = 2*w9*w11 + w11*w12  
ASSERT w14 = w13*w13  
ASSERT w15 = -w8 + 1  
ASSERT w16 = 2*w8*w14 + w14*w15  
ASSERT w17 = w16*w16  
ASSERT w18 = -w7 + 1  
ASSERT w19 = 2*w7*w17 + w17*w18  
ASSERT w20 = w19*w19  
ASSERT w21 = -w6 + 1  
ASSERT w22 = 2*w6*w20 + w20*w21  
ASSERT w23 = w22*w22  
ASSERT w24 = -w5 + 1  
BRILLIG CALL func: 2, inputs: [2*w23*w5 + w23*w24], outputs: [w25]  
ASSERT w26 = 2*w5*w23 + w23*w24  
ASSERT 0 = w25*w26 - 1  
BRILLIG CALL func: 0, inputs: [w0, w26], outputs: [w27, w28]  
BLACKBOX::RANGE input: w27, bits: 64  
BLACKBOX::RANGE input: w28, bits: 64  
ASSERT w29 = w26 - w28 - 1  
BLACKBOX::RANGE input: w29, bits: 64  
ASSERT w28 = -w26*w27 + w0  
ASSERT w2 = w27
```

```
unconstrained func 0: main  
0: 02 = const u32 1  
1: 01 = const u32 32839  
2: 00 = const u32 71  
3: sp[3] = const u32 2  
4: sp[4] = const u32 0  
5: 068 = calldata copy [sp[4]; sp[3]]  
6: 068 = cast 068 to u64  
7: 069 = cast 069 to u64  
8: sp[1] = 068  
9: sp[2] = 069  
10: call 16  
11: call 17  
12: 070 = sp[1]  
13: sp[2] = const u32 70  
14: sp[3] = const u32 1  
15: stop &[sp[2]; sp[3]]  
16: return  
17: call 21  
18: sp[3] = u64 shr sp[1], sp[2]  
19: sp[1] = sp[3]  
20: return  
21: 04 = const u32 30791  
22: 03 = u32 lt 00, 04  
23: jump if 03 to 26  
24: 01 = indirect const u64 15764276373176857197  
25: trap &[01; 02]  
26: return
```

Computation complexity

ACIR is very compute-heavy:

- Finite field logic
- No “quick paths”
- Non-field arithmetic requires lots of operations

Dozens of times slower than brillig for most useful programs

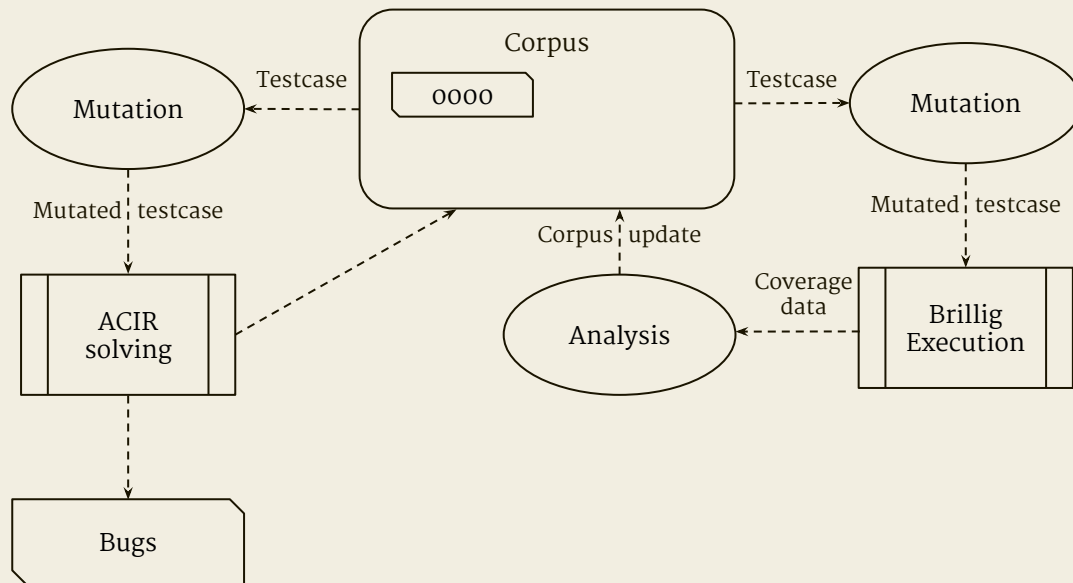
```
func 0
private parameters: [w0]
public parameters: [w1]
return values: [w2]
BLACKBOX:::RANGE input: w0, bits: 64
BLACKBOX:::RANGE input: w1, bits: 64
BRILLIG CALL func: 0, inputs: [w1 + 18446744073709551552,
18446744073709551616], outputs: [w3, w4]
BLACKBOX:::RANGE input: w4, bits: 64
ASSERT w4 = w1 - 18446744073709551616*w3 + 18446744073709551552
ASSERT w3 = 0
BRILLIG CALL func: 1, inputs: [w1, 6, 2], outputs: [[w5, w6, w7,
w8, w9, w10]]
BLACKBOX:::RANGE input: w5, bits: 1
BLACKBOX:::RANGE input: w6, bits: 1
BLACKBOX:::RANGE input: w7, bits: 1
BLACKBOX:::RANGE input: w8, bits: 1
BLACKBOX:::RANGE input: w9, bits: 1
BLACKBOX:::RANGE input: w10, bits: 1
ASSERT w5 = w1 - 2*w6 - 4*w7 - 8*w8 - 16*w9 - 32*w10
ASSERT w11 = w10*w10 + 2*w10 + 1
ASSERT w12 = -w9 + 1
ASSERT w13 = 2*w9*w11 + w11*w12
ASSERT w14 = w13*w13
ASSERT w15 = -w8 + 1
ASSERT w16 = 2*w8*w14 + w14*w15
ASSERT w17 = w16*w16
ASSERT w18 = -w7 + 1
ASSERT w19 = 2*w7*w17 + w17*w18
ASSERT w20 = w19*w19
ASSERT w21 = -w6 + 1
ASSERT w22 = 2*w6*w20 + w20*w21
ASSERT w23 = w22*w22
ASSERT w24 = -w5 + 1
BRILLIG CALL func: 2, inputs: [2*w23*w5 + w23*w24], outputs: [w25]
ASSERT w26 = 2*w5*w23 + w23*w24
ASSERT 0 = w25*w26 - 1
BRILLIG CALL func: 0, inputs: [w0, w26], outputs: [w27, w28]
BLACKBOX:::RANGE input: w27, bits: 64
BLACKBOX:::RANGE input: w28, bits: 64
ASSERT w29 = w26 - w28 - 1
BLACKBOX:::RANGE input: w29, bits: 64
ASSERT w28 = -w26*w27 + w0
ASSERT w2 = w27
```

Solution

Separate the fuzzing loops:

1. Two independent fuzzers share the same corpus
2. Time is distributed equally

Results in faster feature discovery



What do we do with code like this?

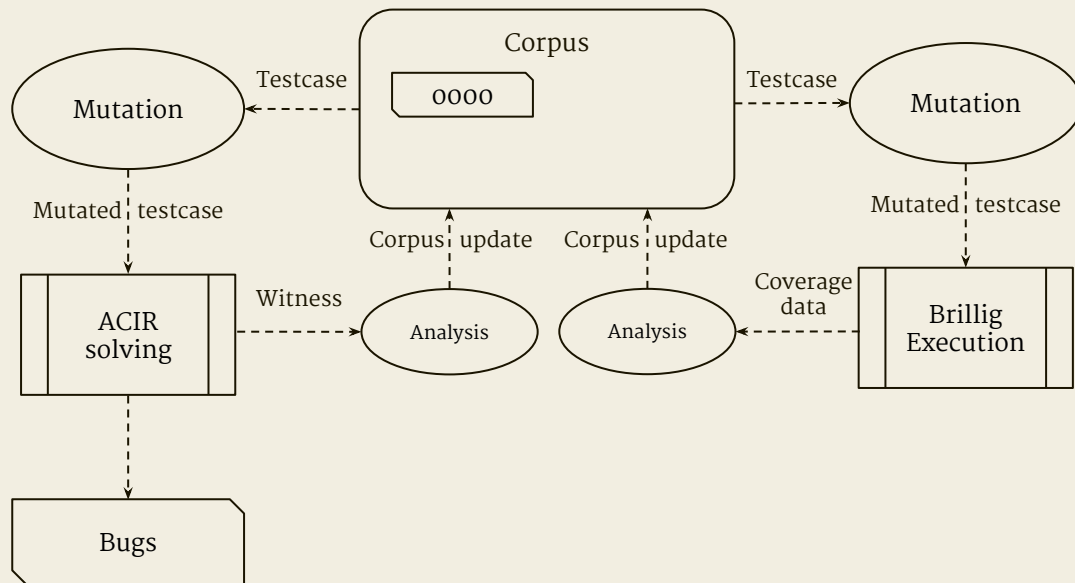
```
fn main(secret: [u8; 10]) {  
    let key = "POWER_OF_COMMUNITY".as_bytes();  
    let result = "FUZZING_IS_AWESOME".as_bytes();  
    let mut safisfied = true;  
    for i in 0..10 {  
        safisfied = safisfied & (secret[i] ^ key[i] == result[i]); // No branching :(  
    }  
  
    assert(!safisfied);  
}
```

Solution

ACIR variables that deal with logic only have boolean values

So we add witness analysis:

1. Parse witness
2. Select values that are only 0 or 1
3. Treat as additional coverage
4. If a variable suddenly get a value different from 0 or 1, stop tracking



Demo



Let's fuzz this

```
#[fuzz]
fn fuzz_add(a: u64, b: u64) {
    assert((a + b - 16) != (a - b + 30));
}
```

Demo



Circuit caveats

A lot of failures are intended and the fuzzer will constantly run into them

We need to be able to specify interesting conditions

```
Failed assertion: attempt to subtract with overflow
Failing input:
{"a":"0x00","b":"0x00"}
saved input to:
"/mnt/user-data/kesha/Development/Noir/test_programs/fuzz_test/Prov
error: Assertion failed: attempt to subtract with overflow
  src/main.nr:68:13
68 |         assert((a + b - 16) != (a - b + 30), "message");
    |         ~~~~~
= Call stack:
  1. src/main.nr:68:13

[fuzz_test] fuzz_add failed
% exit
```

Failure selection

```
#[fuzz(only_fail_with = "message")]  
fn fuzz_add(a: u64, b: u64) {  
    assert((a + b - 16) != (a - b + 30), "message");  
}
```

Demo



Additional features

- Multithreading
- Time / count limits
- Fuzz tests for CI integration
- Specialized mutations for each basic type
- Automatic structured mutations based on input structure
- Testcase splicing
- Autodict

```
struct A {  
    a: bool,  
}  
struct B {  
    a: A,  
    b: bool,  
}  
struct TestStruct {  
    a: A,  
    b: B,  
    d: u32,  
}  
#[fuzz(only_fail_with = "T_T")]  
fn fuzz_test_struct(t_struct: TestStruct) {  
    assert(  
        !(t_struct.a.a & !t_struct.b.b & t_struct.b.a.a & (t_struct.d  
+ 0xbaddc0de == 0xdeadbeef)),  
        "T_T",  
    );  
}
```

Fuzzing oracle integration

Developers can implement JsonRPC oracles to compare to a different implementation

```
from werkzeug.wrappers import Request, Response
from werkzeug.serving import run_simple

from jsonrpc import JSONRPCResponseManager, dispatcher

from json import loads, JSONDecodeError

@dispatcher.add_method
def resolve_foreign_call(arg):
    assert arg["function"]=="correct_json"
    text=bytes([int(x,16) for x in arg["inputs"]][0])
    result = dict()
    success=True
    try:
        decoded=loads(text)
    except (JSONDecodeError, UnicodeDecodeError):
        success=False
    result['values']=["1" if success else "0"]
    return result

@Request.application
def application(request):
    response = JSONRPCResponseManager.handle(
        request.data, dispatcher)
    return Response(response.json,
        mimetype='application/json')

if __name__ == '__main__':
    run_simple('localhost', 40000, application)
```

JSON library bug discovery demo

```
#[fuzz(only_fail_with = "incorrect_json")]
fn main_str_shortest(text: str<32>) {
    // If this fails we won't reach the assert
    let mut final_text: [u8; 256] = [32; 256];
    let text_bytes = text.as_bytes();
    for i in 0..32 {
        final_text[256 - 32 - 1 + i] = text_bytes[i];
    }
    final_text[256 - 32 - 2] = "{".as_bytes()[0];
    final_text[255] = "}".as_bytes()[0];
    let mut final_final_text: [u8; 512] = [32; 512];
    for i in 0..256 {
        final_final_text[i + 256] = final_text[i];
    }
    let json = JSON512b::parse_json(final_final_text);
    assert(unsafe { check_json(final_text) },
        "incorrect_json");
}
```

Thank you