香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

NDSS SYMPOSIUM/2025

MTZK: Testing and Exploring Bugs in Zero-Knowledge (ZK) Compilers

NDSS Symposium 2025

Dongwei Xiao, Zhibo Liu, Yiteng Peng and Shuai Wang

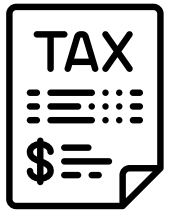
The Hong Kong University of Science and Technology

Are You Concerned About…?

Income > 2K

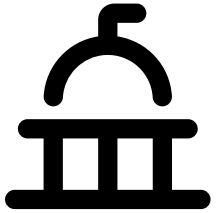
Salary slips?

Bank



Regulatory compliance

Business details?



Government

Carbon neutral

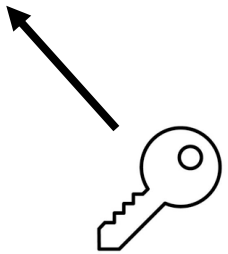
Industrial Data?

Global organization

Zero-Knowledge (ZK) Proofs

Verifier V

Prover P



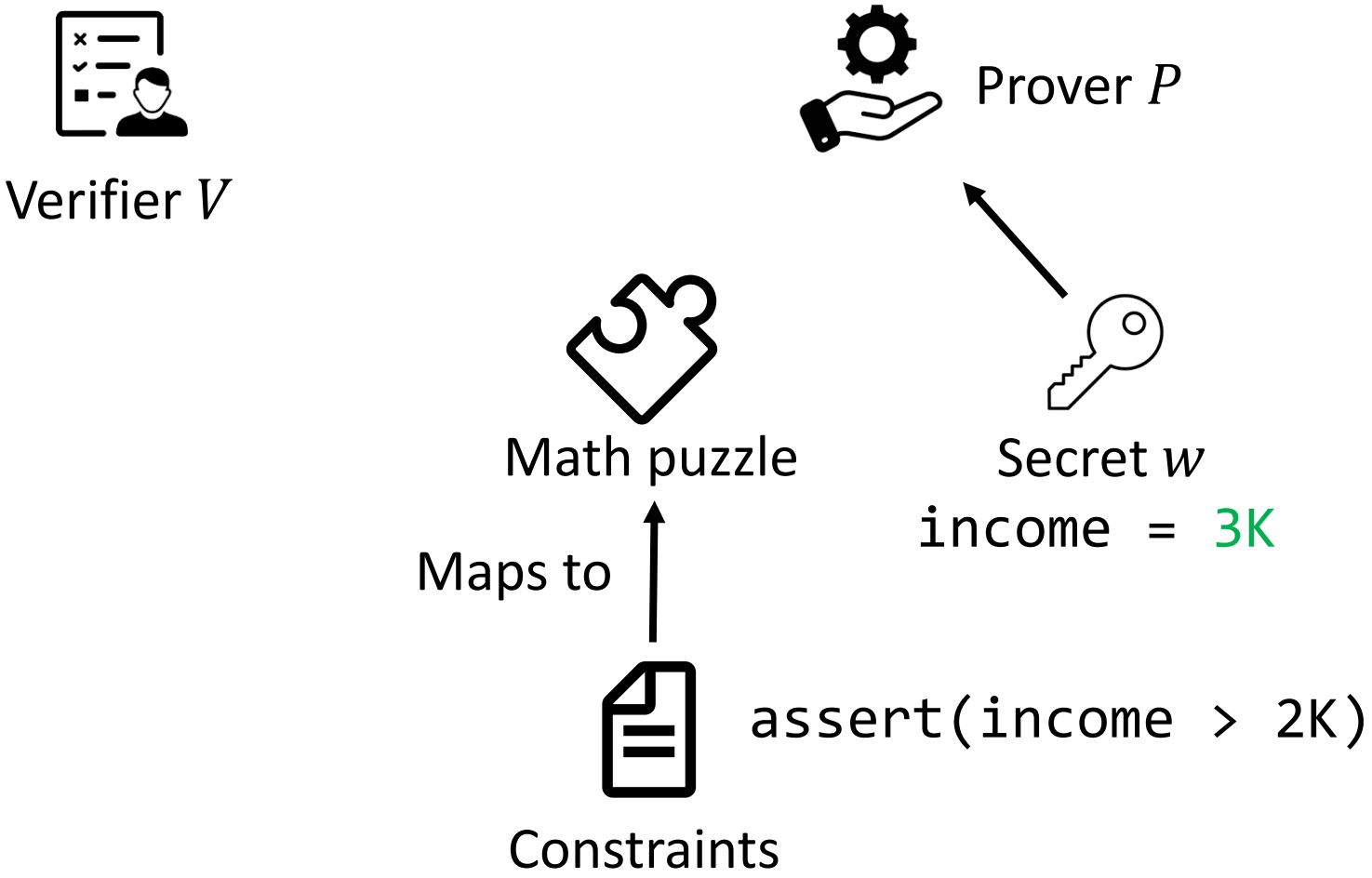
Secret w

```
income = 3K
```

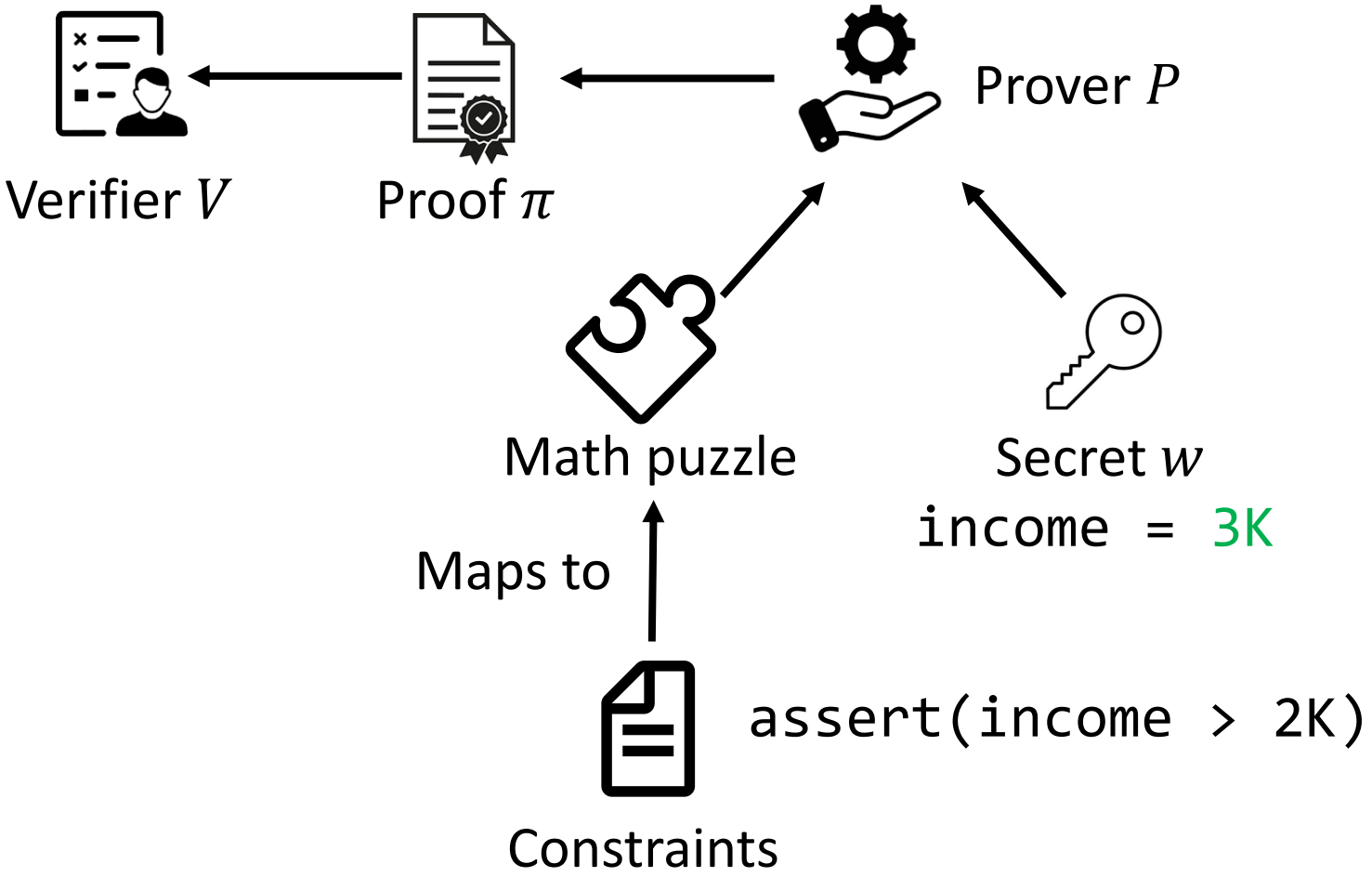
```
assert(income > 2K)
```

Constraints

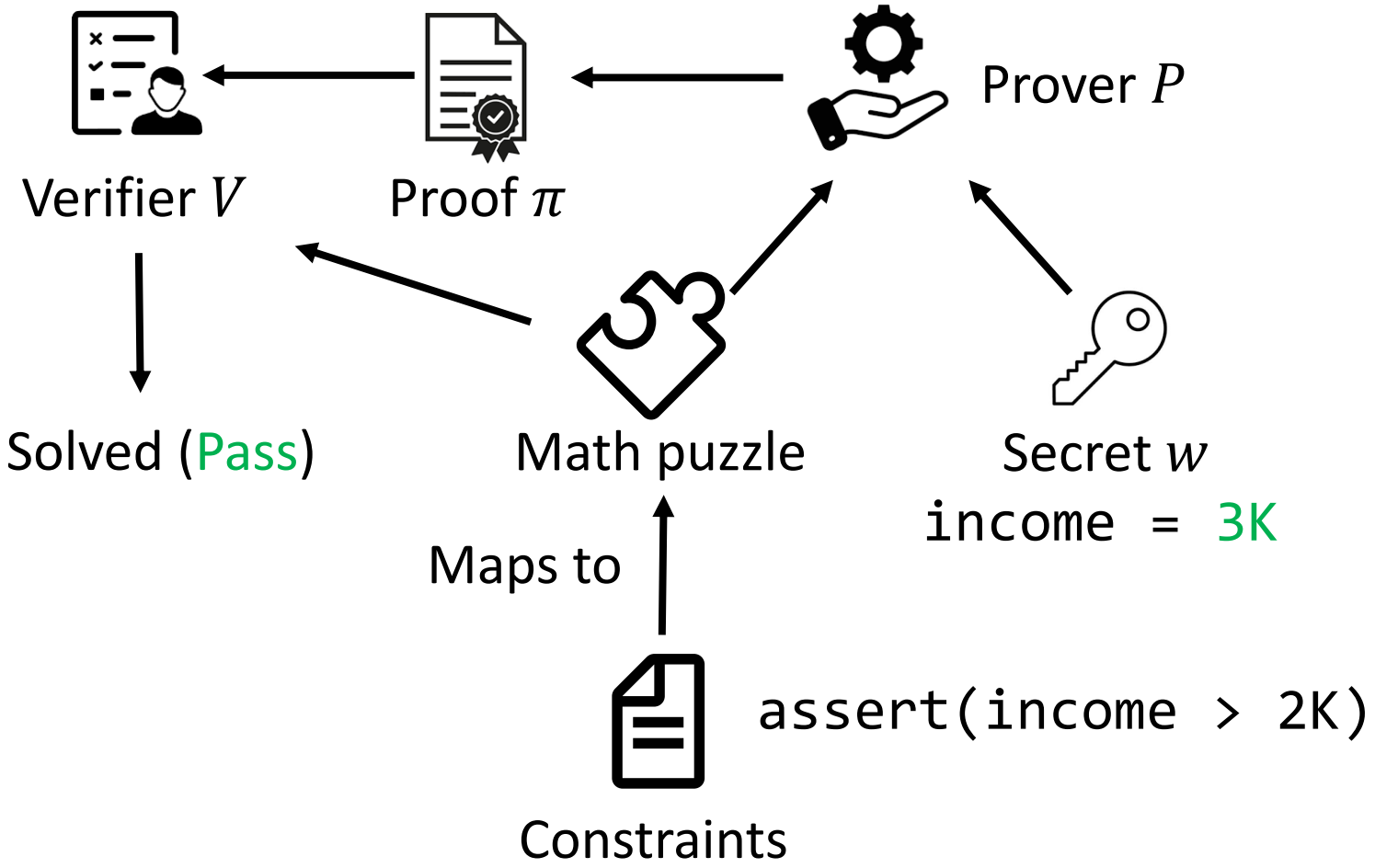
Zero-Knowledge (ZK) Proofs



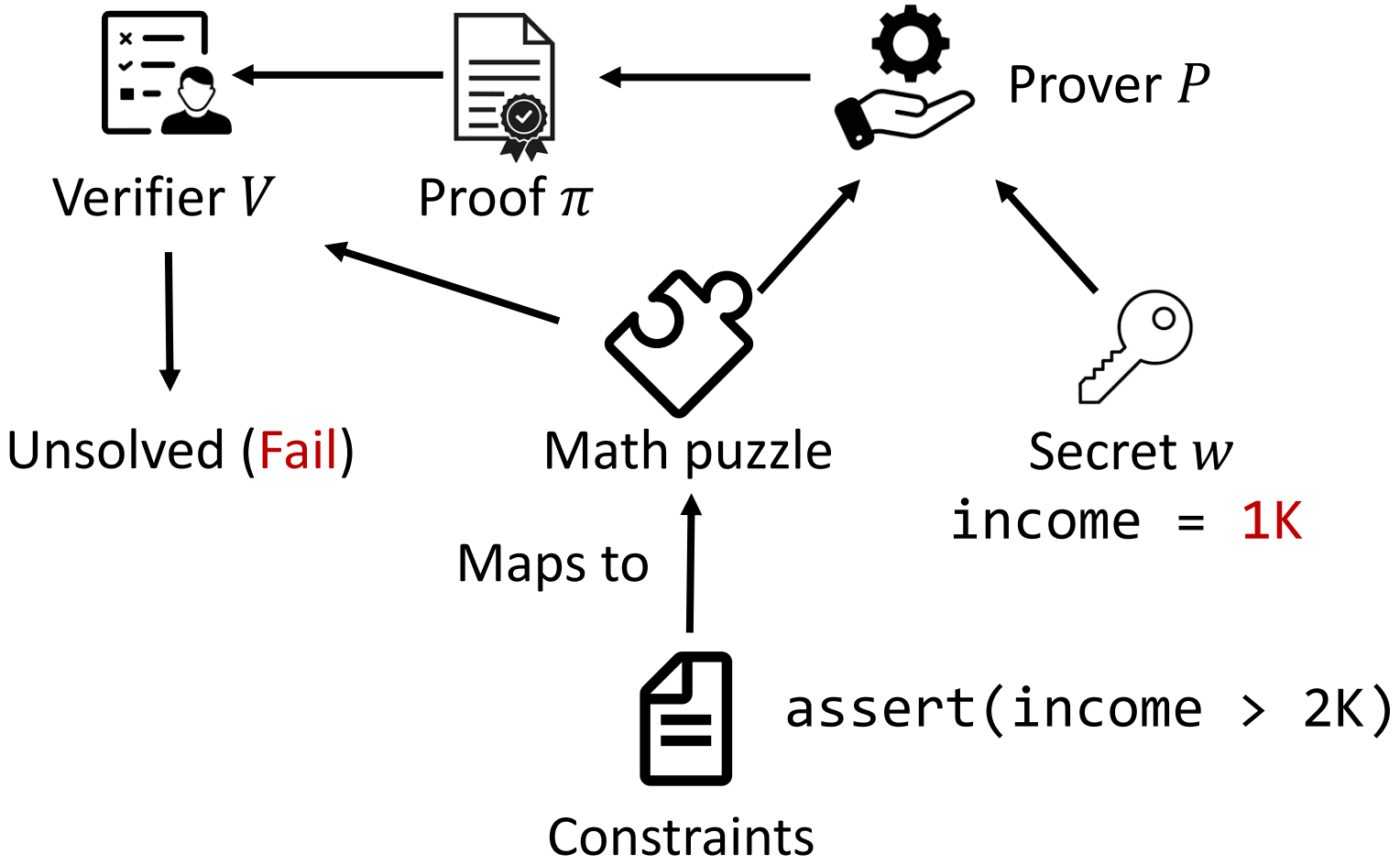
Zero-Knowledge (ZK) Proofs



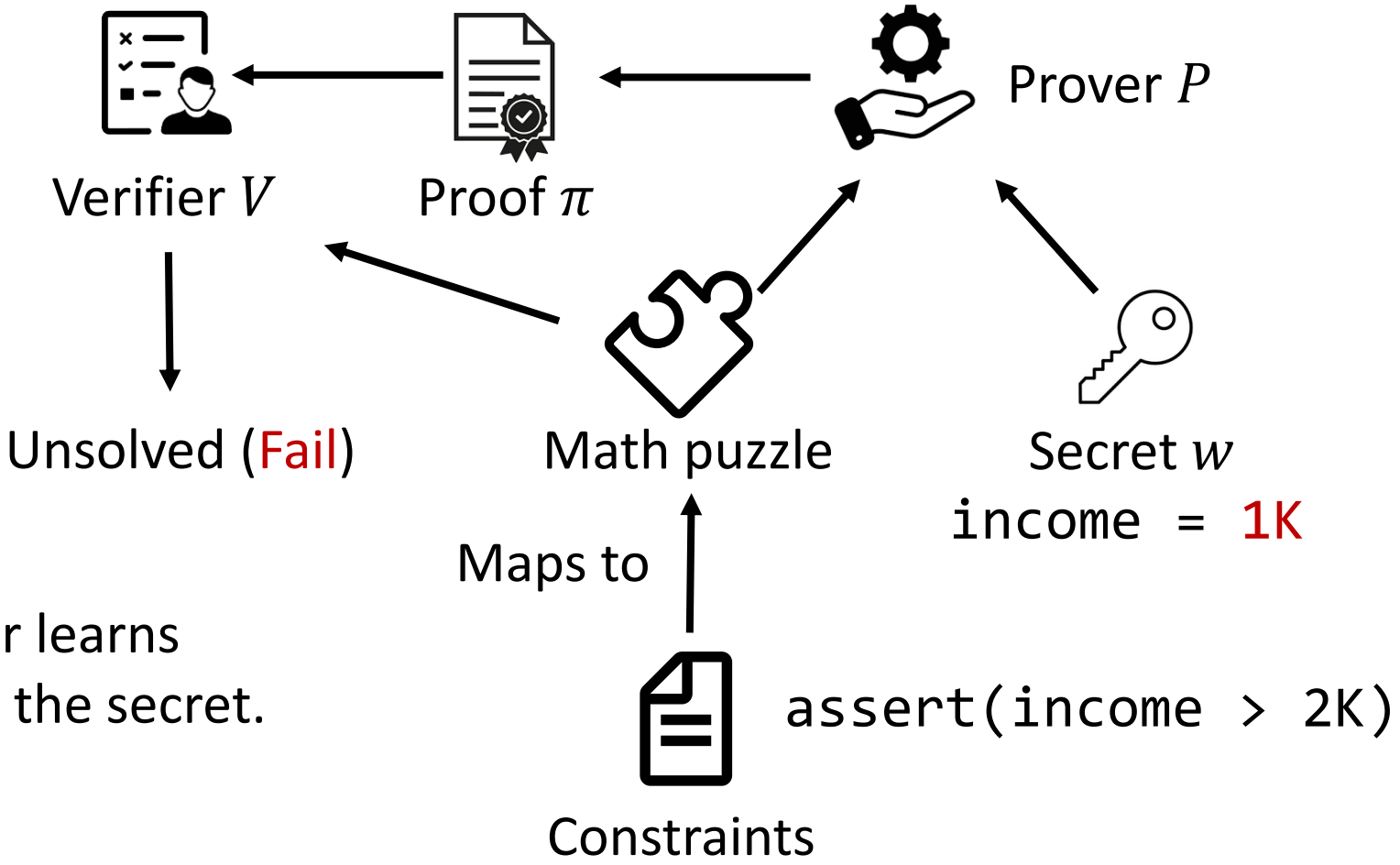
Zero-Knowledge (ZK) Proofs



Zero-Knowledge (ZK) Proofs

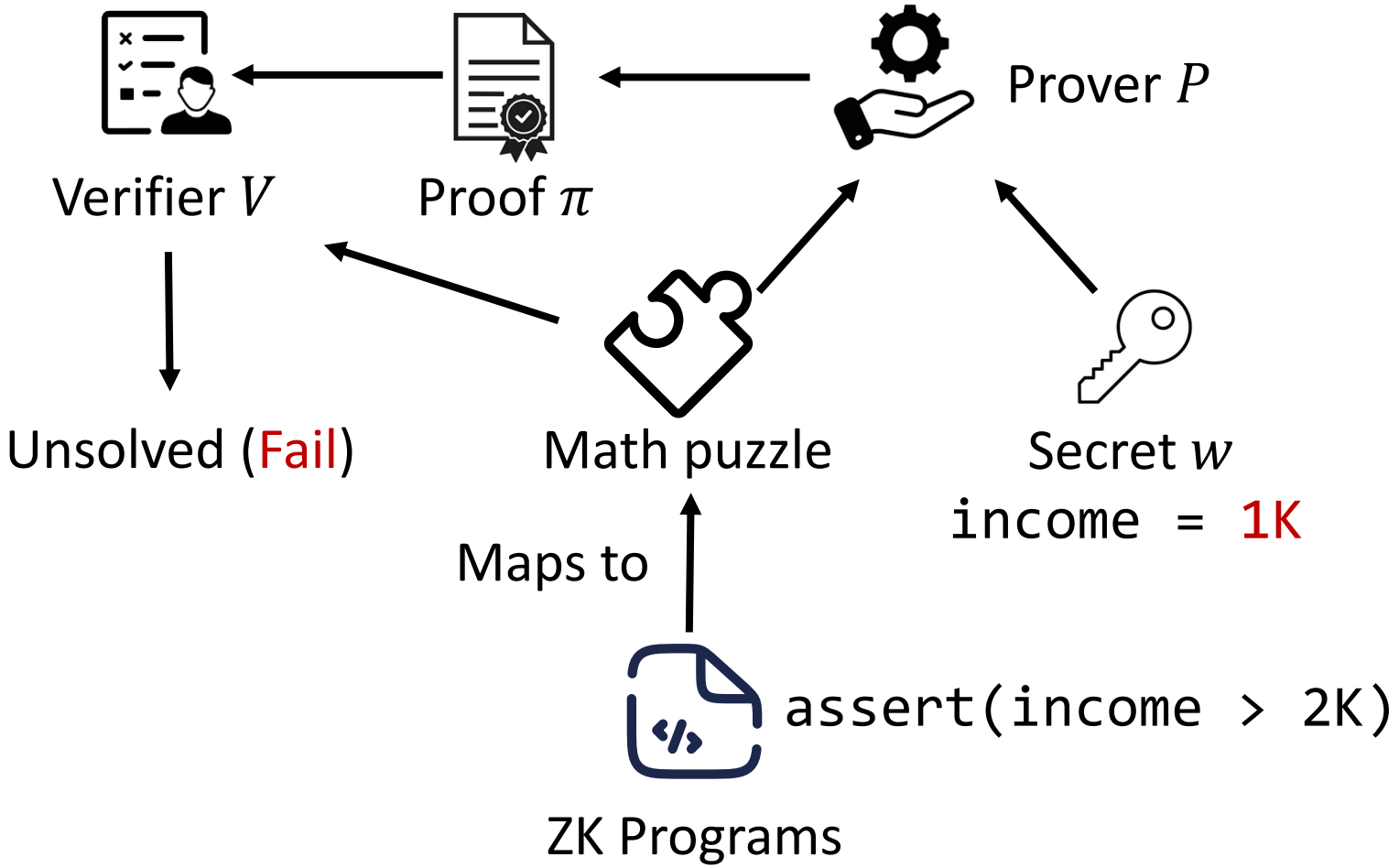


Zero-Knowledge (ZK) Proofs



ZK: The verifier learns nothing about the secret.

Zero-Knowledge (ZK) Proofs



ZK Domain-Specific Language (DSL)

```
def main(public int tax_allowance,
         private int gross_income):
  int taxable_income = gross_income -
                       tax_allowance
  if (taxable_income < 0)
    taxable_income = 0
  int tax = taxable_income >> 2
  int net_income = gross_income - tax
  assert(net_income >= 2000)
```

⇦ Tax rate is 25% (right-shift by 2)

A ZK program checking whether net income >= 2K

ZK Domain-Specific Language (DSL)

```
def main(public int tax_allowance,
         private int gross_income):
  int taxable_income = gross_income -
                       tax_allowance
  if (taxable_income < 0)
    taxable_income = 0
  int tax = taxable_income >> 2
  int net_income = gross_income - tax
  assert(net_income >= 2000)
```

- Visibility modifiers:
  - Public: visible to both prover and verifier

A ZK program checking whether net income >= 2K

ZK Domain-Specific Language (DSL)

```
def main(public int tax_allowance,
         private int gross_income):
  int taxable_income = gross_income -
                       tax_allowance
  if (taxable_income < 0)
    taxable_income = 0
  int tax = taxable_income >> 2
  int net_income = gross_income - tax
  assert(net_income >= 2000)
```

- Visibility modifiers:
  - Public: visible to both prover and verifier
  - Private: visible only to the prover

A ZK program checking whether net income >= 2K

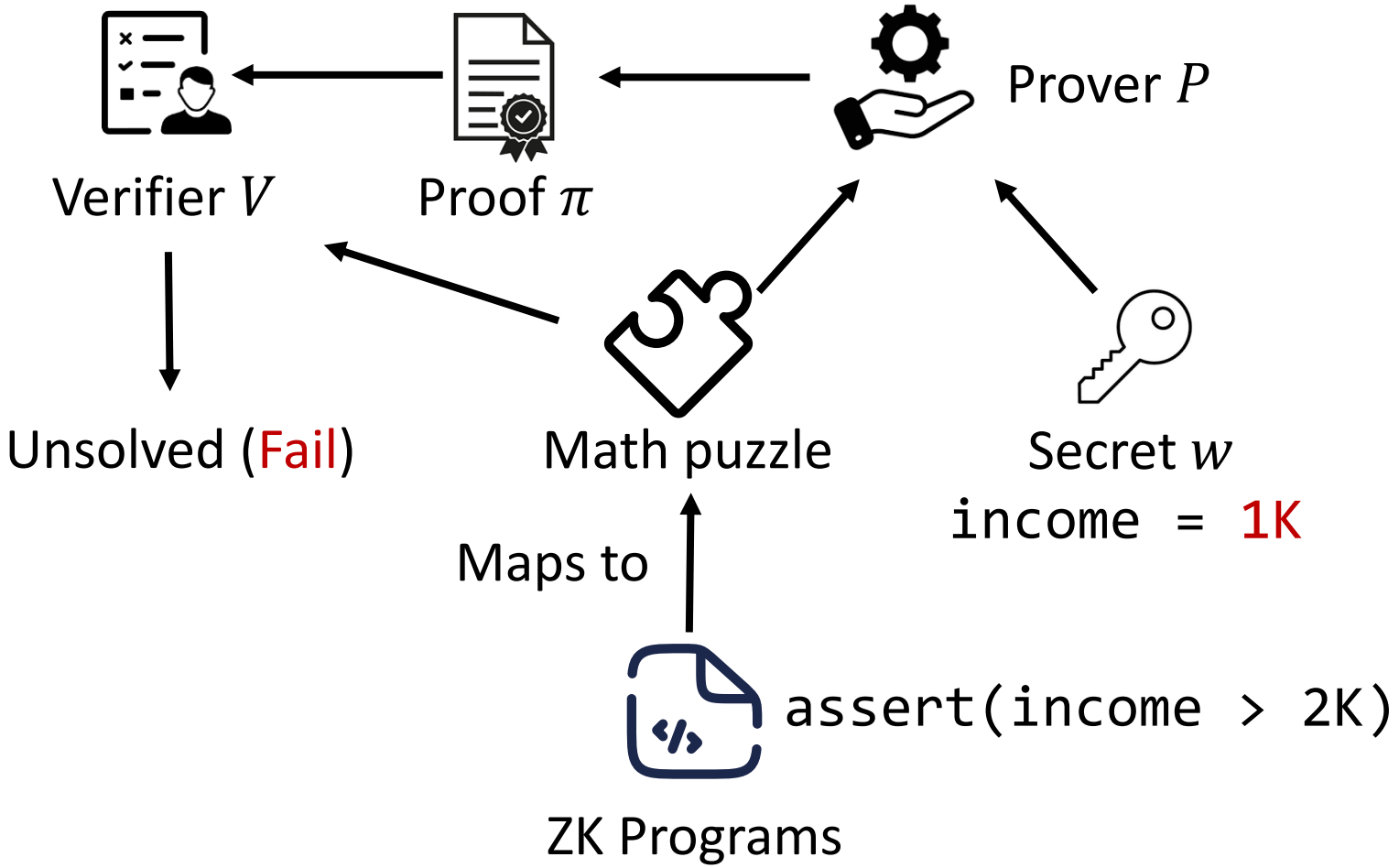
ZK Domain-Specific Language (DSL)

```
def main(public int tax_allowance,
         private int gross_income):
  int taxable_income = gross_income -
                       tax_allowance
  if (taxable_income < 0)
    taxable_income = 0
  int tax = taxable_income >> 2
  int net_income = gross_income - tax
  assert(net_income >= 2000)
```

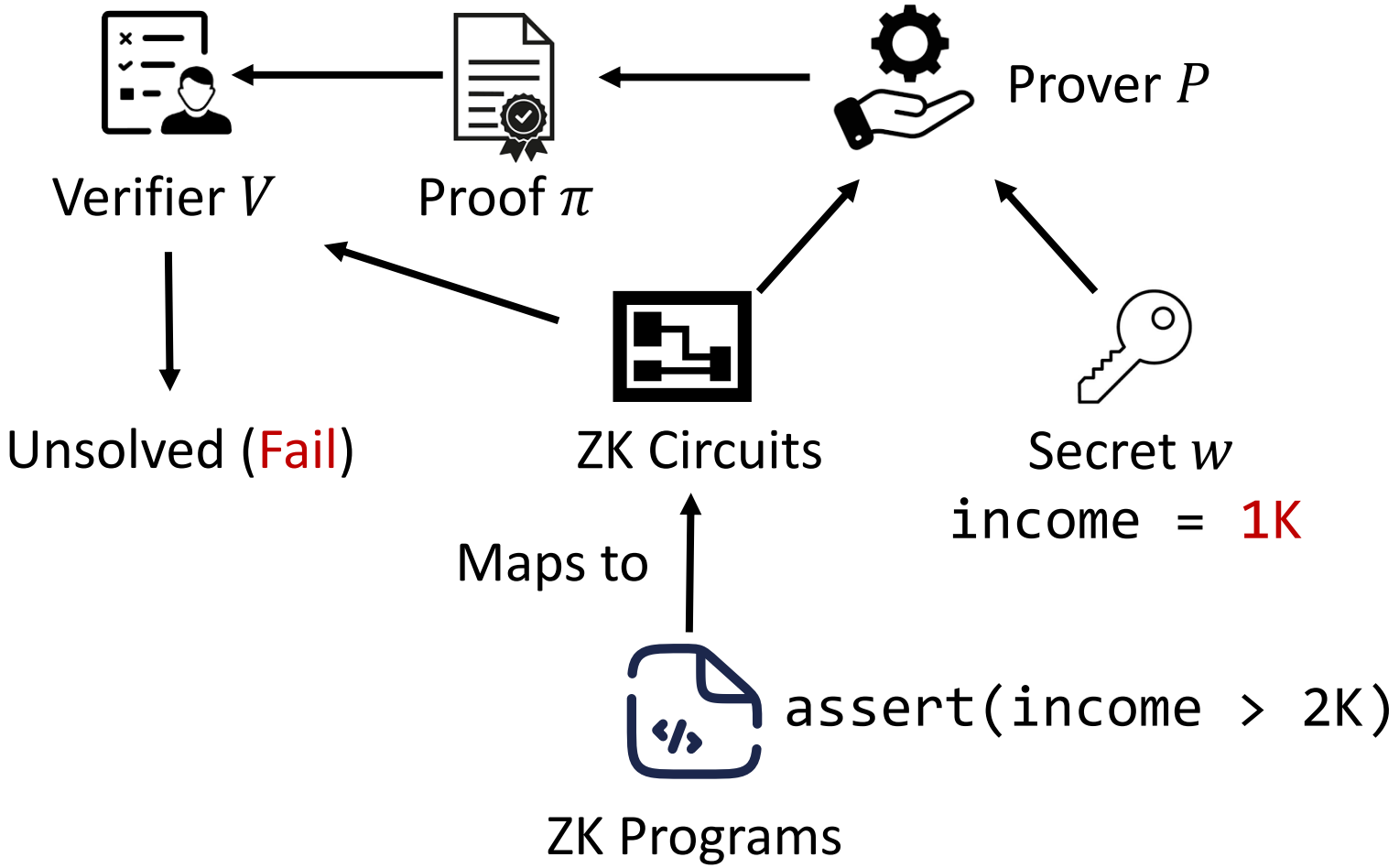
- Visibility modifiers:
  - Public: visible to both prover and verifier
  - Private: visible only to the prover
- Constraints to check

A ZK program checking whether net income >= 2K

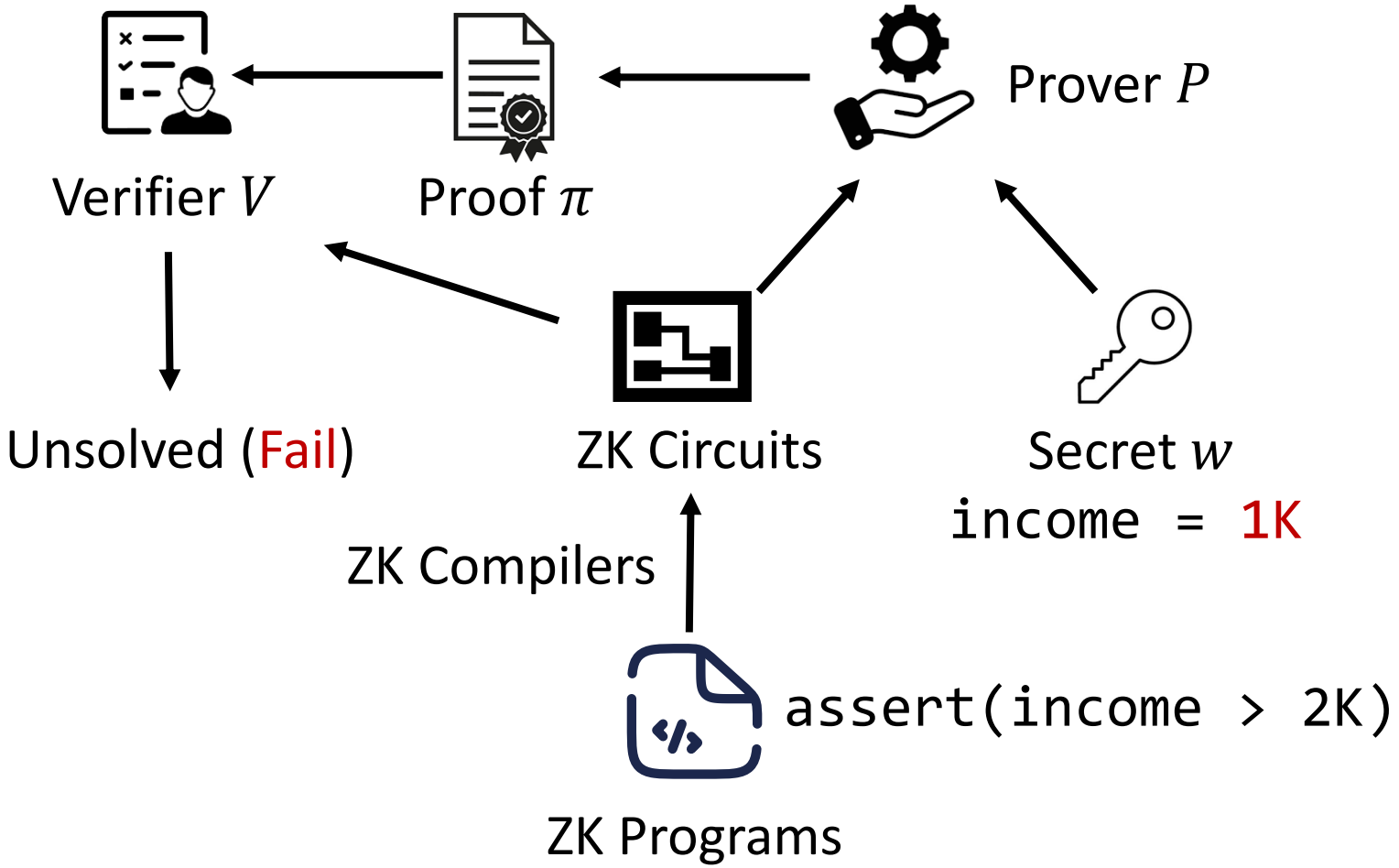
Zero-Knowledge (ZK) Proofs



Zero-Knowledge (ZK) Proofs

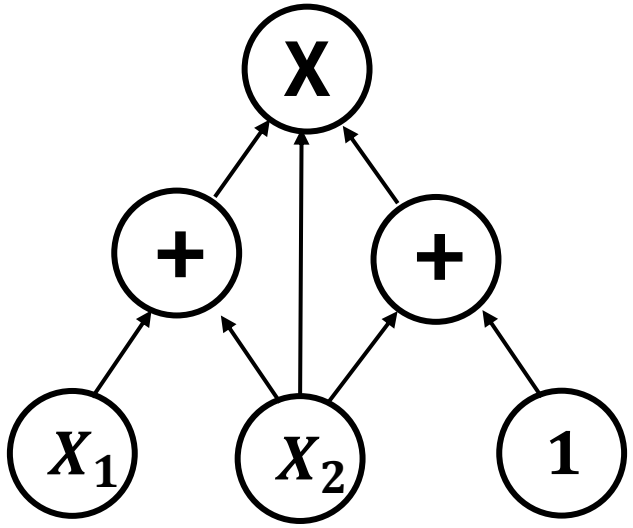


Zero-Knowledge (ZK) Proofs



ZK Circuits

- No control flow
- Only addition and multiplication
- Represent crypto primitives
- Everything is bits



ZK Compilers

- Flatten If, For, While statements
- Arithmetization
- Crypto-aware optimizations
- Convert all data types to bits

Flattener

Visibility Checker

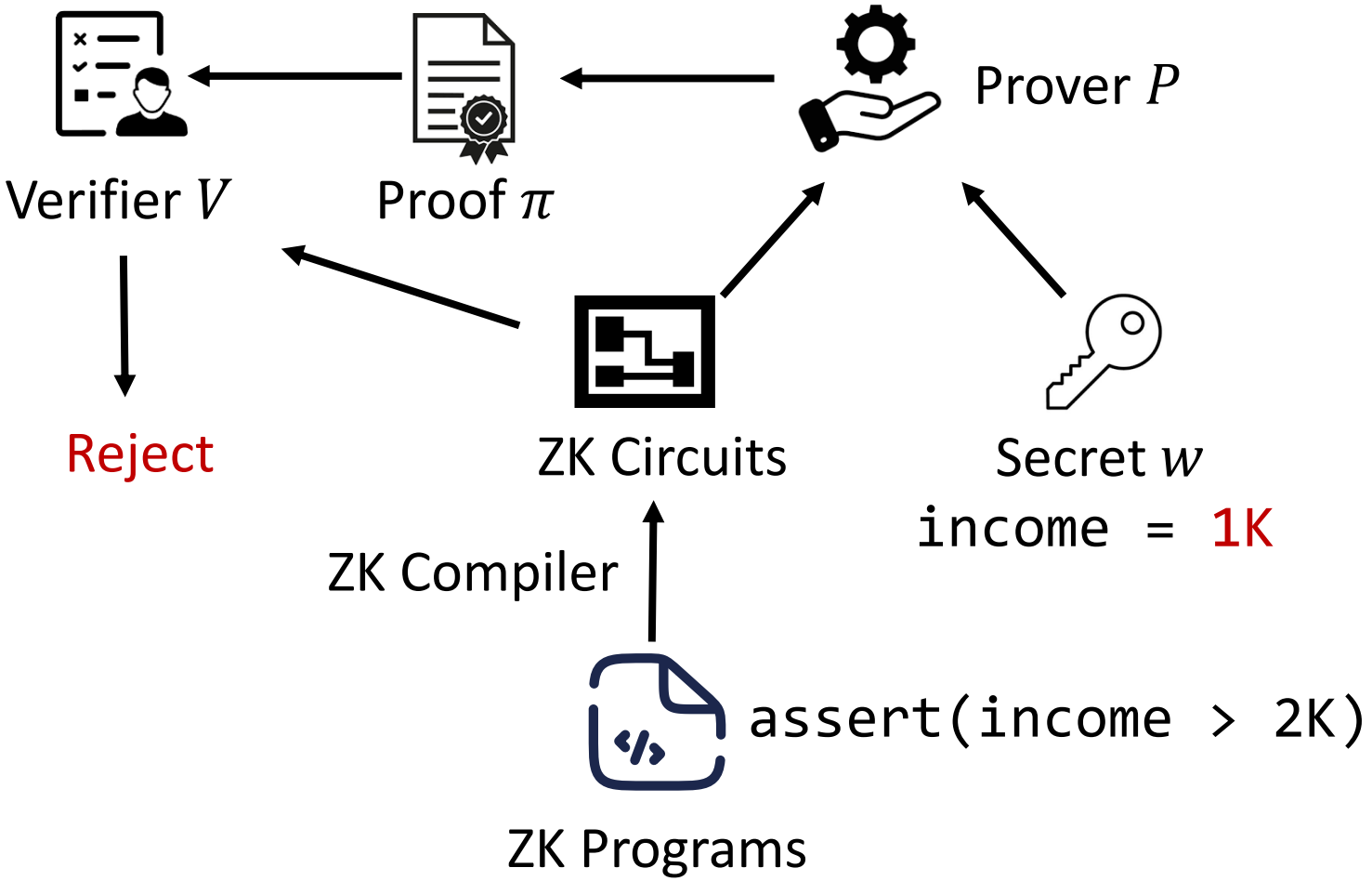
Front-end Optimizer

Lowering

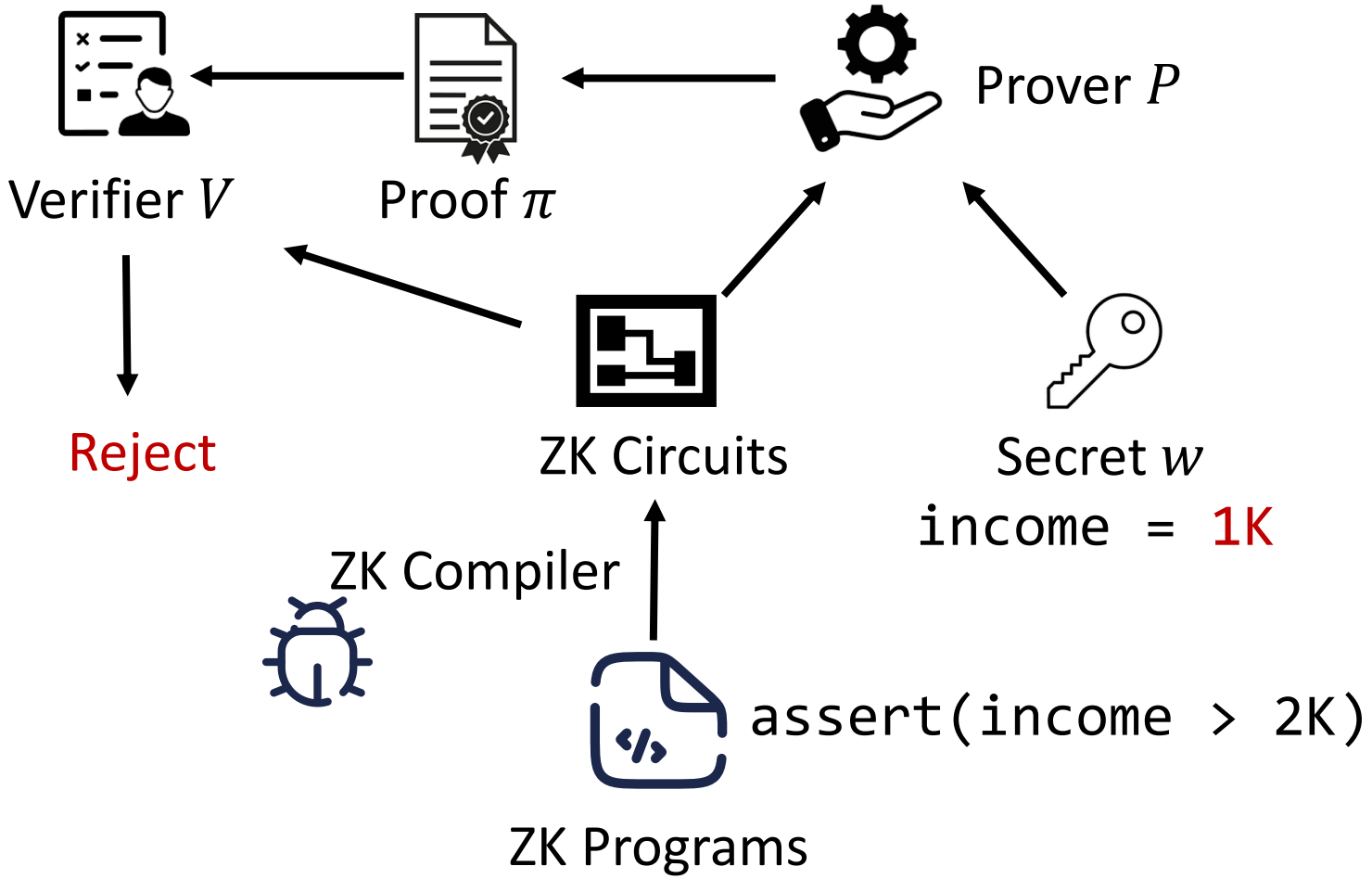
Arithme-tization

Back-End Optimizer

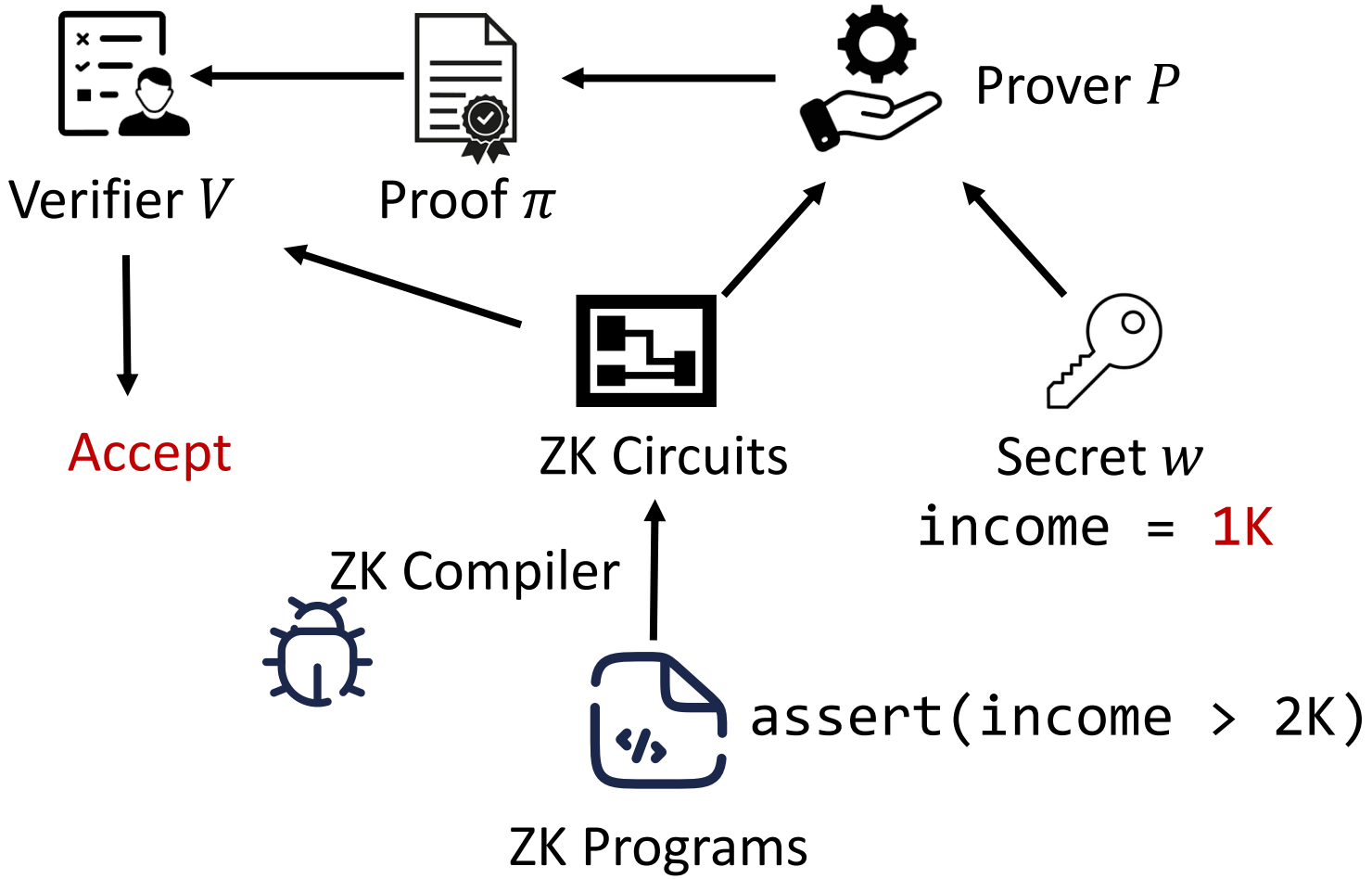
Correct Compilation Refuses the Unqualified



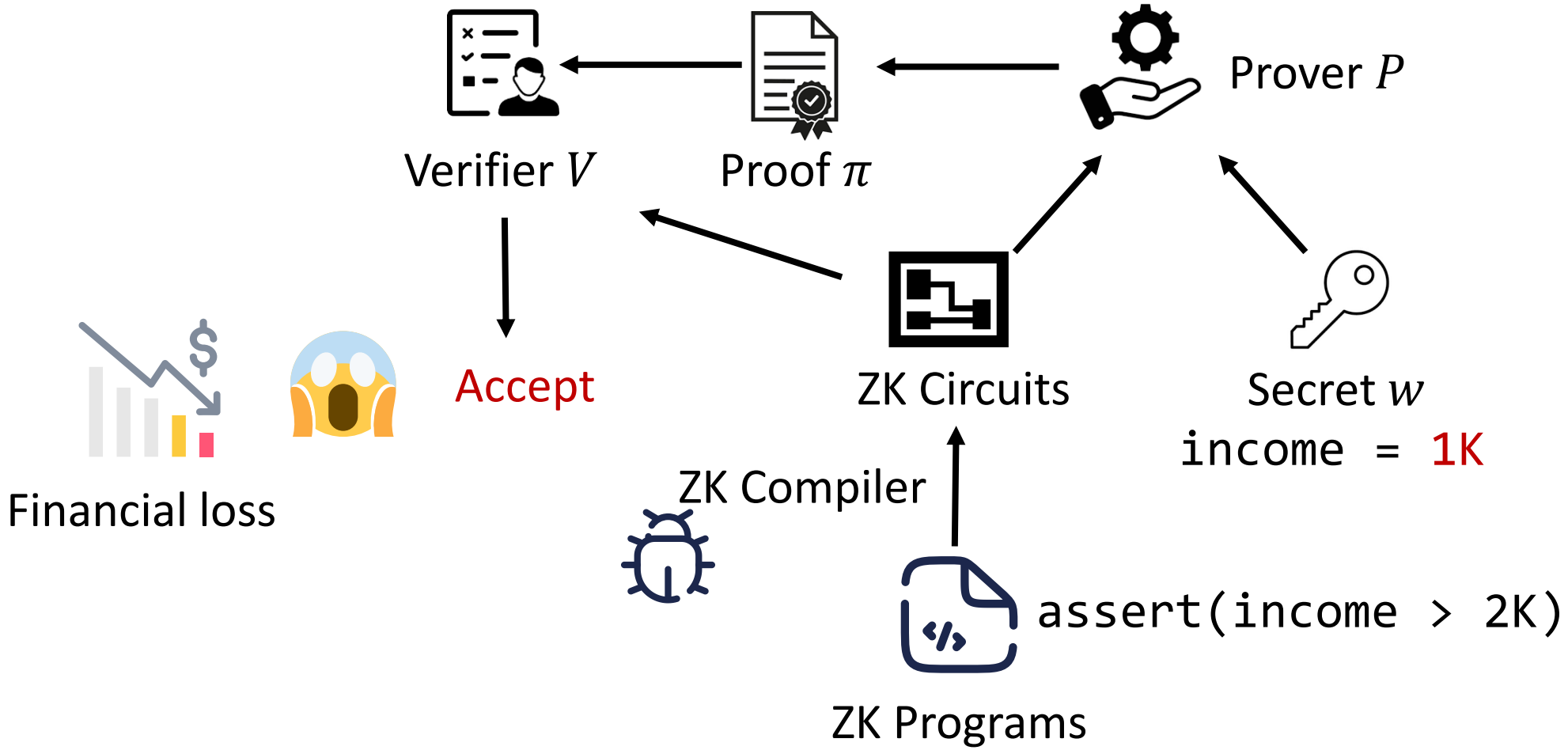
Erroneous ZK Compilers



Erroneous ZK Compilers



Erroneous ZK Compilers



Real-world ZK Bugs Affecting Blockchains

1. Lack of range constraints for the `tree_index` variable

Description of bug:

The index (i.e. position) of a note in the Aztec 2.0 tree was used in the computation of the note nullifier. The code assumed this to be a 32-bit integer, and so used only the last 32-bits of this element as the tree index. However, the field element representing this position was not actually constrained to be 32 bits. Therefore, the entire field element was used as an input for computing the nullifier.

Summary

During the final activation phase of the Aave v3 ZKSync pool on August 21th [36], a problem with its behaviour was detected. Consequently, a global pause of all assets was enacted by the Aave Guardian, delaying the activation of the new instance.

The company behind the privacy-minded cryptocurrency zcash has revealed that it fixed a catastrophic code bug last year that could have been used to print infinite coins.

https://governance.aave.com/t/bgd-aave-v3-zksync-activation-issue-report/18819
https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities
https://www.coindesk.com/markets/2019/02/05/zcash-team-reveals-it-fixed-a-catastrophic-coin-counterfeiting-bug

Real-world ZK Bugs Affecting Blockchains

1. Lack of range constraints for the `tree_index` variable

Description of bug:

The index (i.e. position) of a note in the Aztec 2.0 tree was used in the computation of the note nullifier. The code assumed this to be a 32-bit integer, and so used only the last 32-bits of this element as the tree index. However, the field element representing this position was not actually constrained to be 32 bits. Therefore, the entire field element was used as an input for computing the nullifier.

Summary

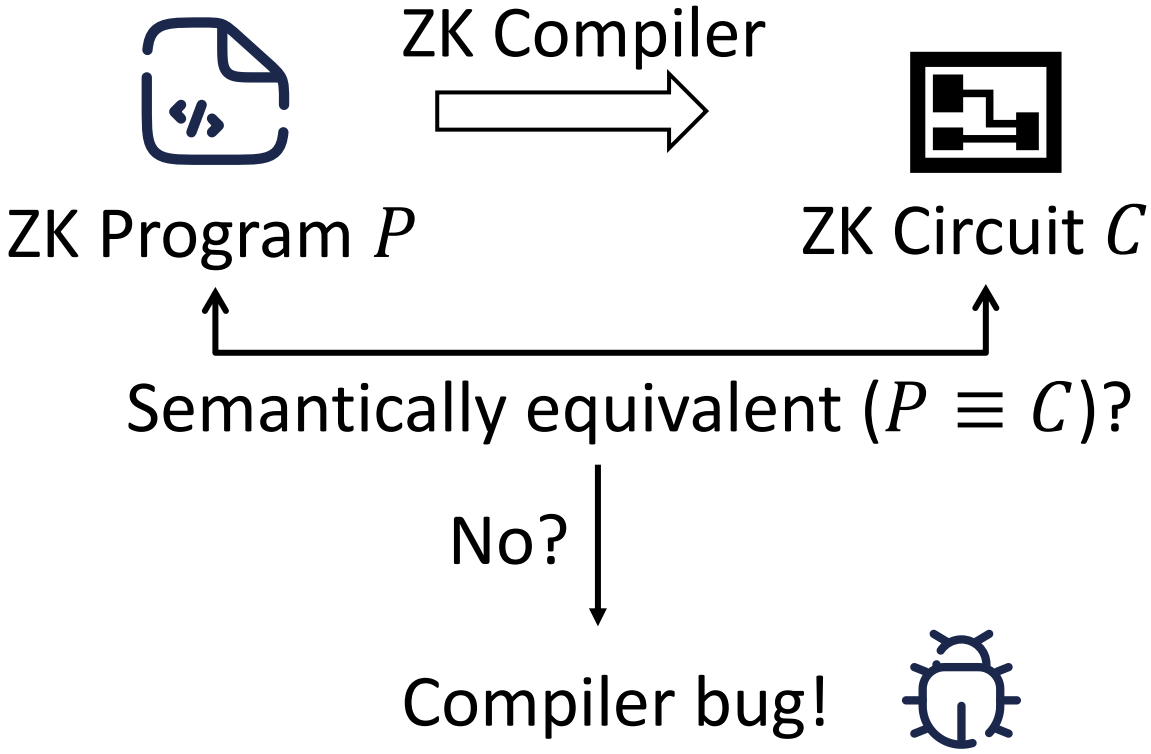
During the final activation phase of the Aave v3 ZKSync pool on August 21th [36], a problem with its behaviour was detected. Consequently, a global pause of all assets was enacted by the Aave Guardian, delaying the activation of the new instance.

The company behind the privacy-minded cryptocurrency zcash has revealed that it fixed a catastrophic code bug last year that could have been used to print infinite coins.

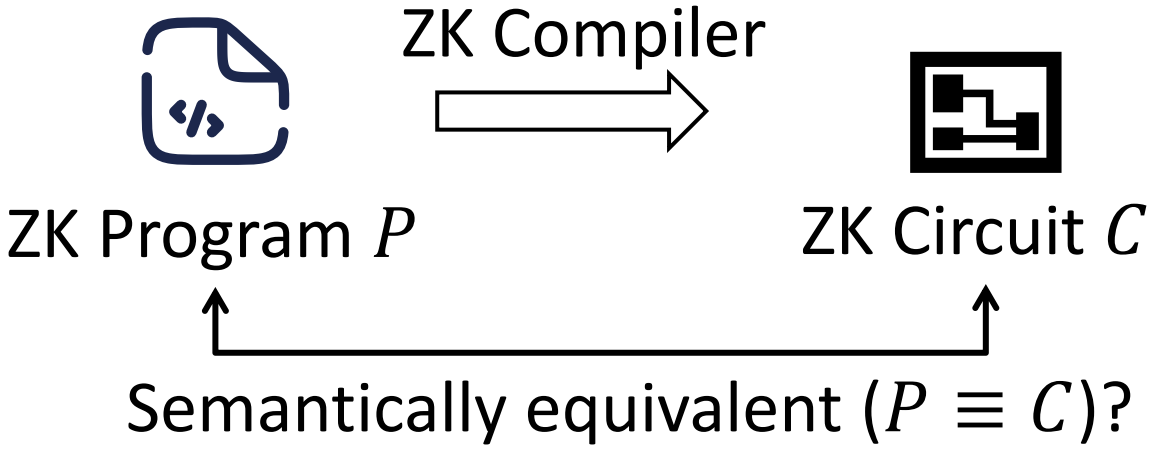
How can we automatically test ZK compilers?

https://governance.aave.com/t/bgd-aave-v3-zksync-activation-issue-report/18819
https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities
https://www.coindesk.com/markets/2019/02/05/zcash-team-reveals-it-fixed-a-catastrophic-coin-counterfeiting-bug

Testing Oracle (Ground Truth) Problem



Testing Oracle (Ground Truth) Problem



Hard!

Metamorphic Mutations

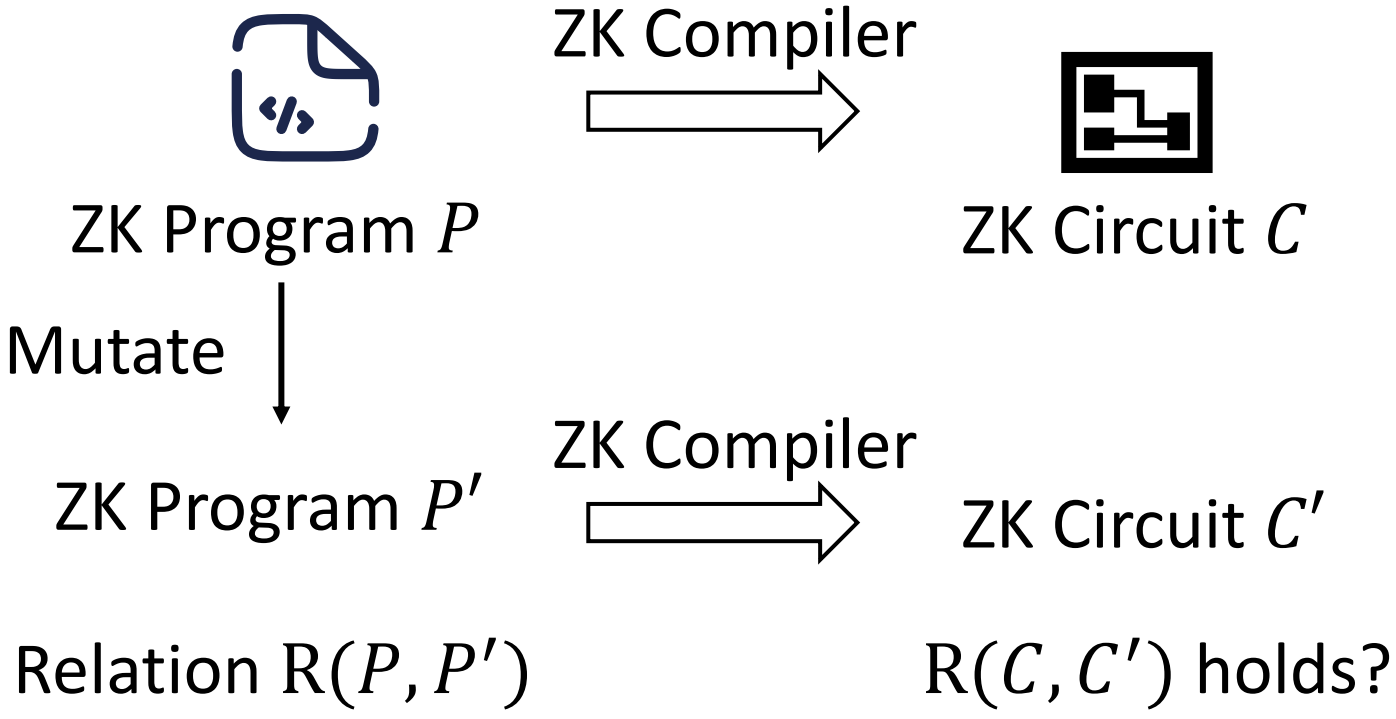
ZK Program P

Mutate

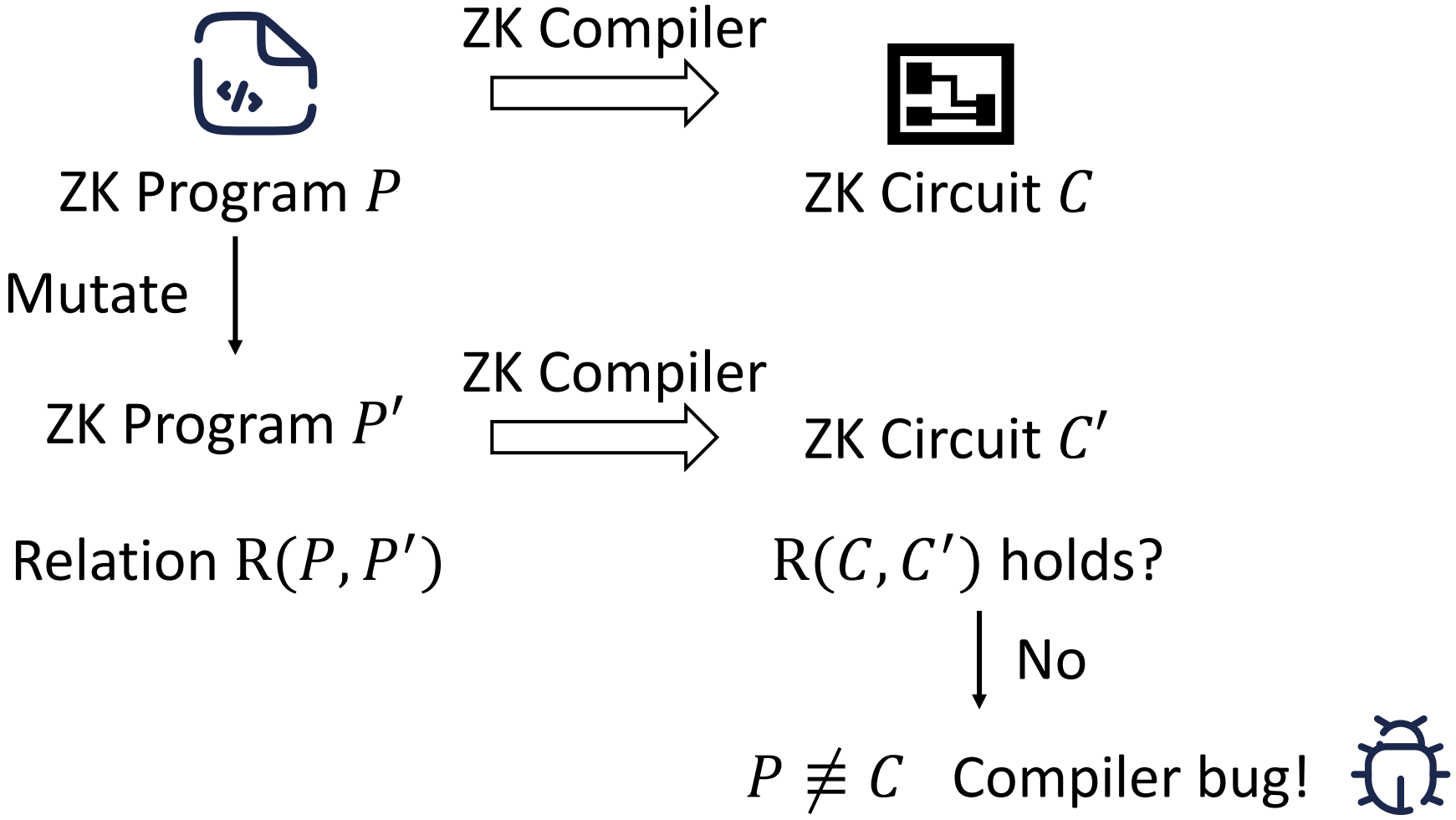
ZK Program P'

Relation $\mathrm{R}(P, P')$

Metamorphic Mutations



Metamorphic Mutations



MR_{SIM}: Satisfiability-Invariant Mutation

ZK Program P

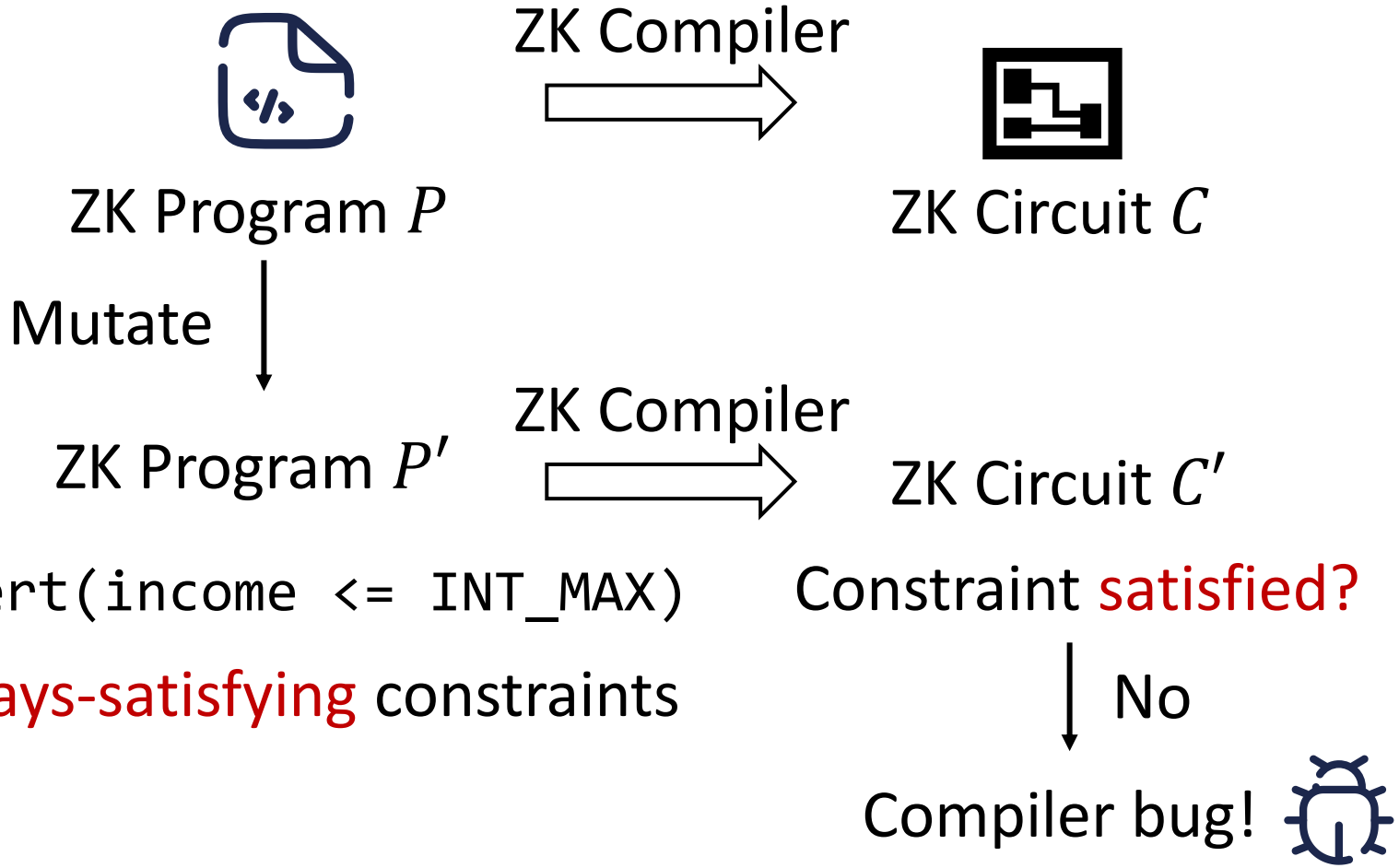
Mutate

ZK Program P'

```
assert(income <= INT_MAX)
```

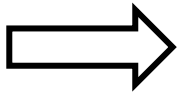
Always-satisfying constraints

MR_{SIM}: Satisfiability-Invariant Mutation



Implementation of MR_{SIM}

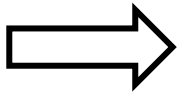
```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6      } else {
7        x -= 1;
8      }
9  }
```



```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6        assert(x <= INT_MAX) // C-MAX
7      } else {
8        x -= 1;
9      }
10 }
```

Implementation of MR_{SIM}

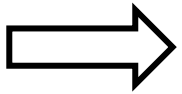
```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6      } else {
7        x -= 1;
8      }
9  }
```



```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6        assert(x <= INT_MAX) // C-MAX
7      } else {
8        x -= 1;
9        assert(x == x * 1)  // C-TAUT
10     }
11 }
```

Implementation of MR_{SIM}

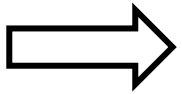
```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6      } else {
7        x -= 1;
8      }
9  }
```



```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6        assert(x <= INT_MAX) // C-MAX
7      } else {
8        x -= 1;
9        assert(x == x * 1)  // C-TAUT
10     }
11   //       C-EQ
12   assert(x == 76)
13 }
```

Implementation of MR_{SIM}

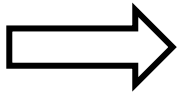
```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6      } else {
7        x -= 1;
8      }
9  }
```



```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6        assert(x <= INT_MAX) // C-MAX
7      } else {
8        x -= 1;
9        assert(x == x * 1)  // C-TAUT
10     }
11   //       C-EQ          C-NLT
12   assert(x == 76     !(x < 76))
13 }
```

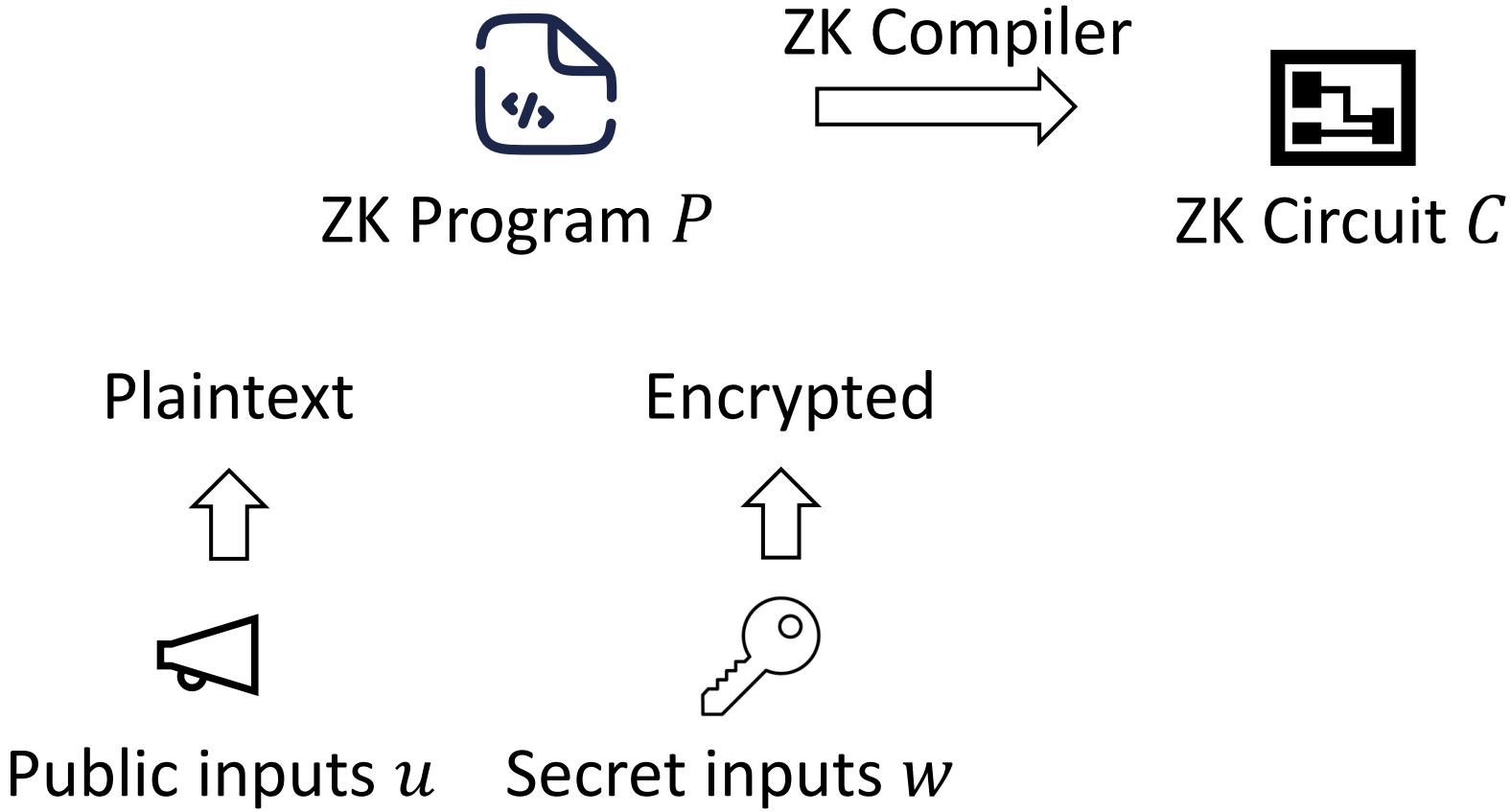
Implementation of MR_{SIM}

```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6      } else {
7        x -= 1;
8      }
9  }
```

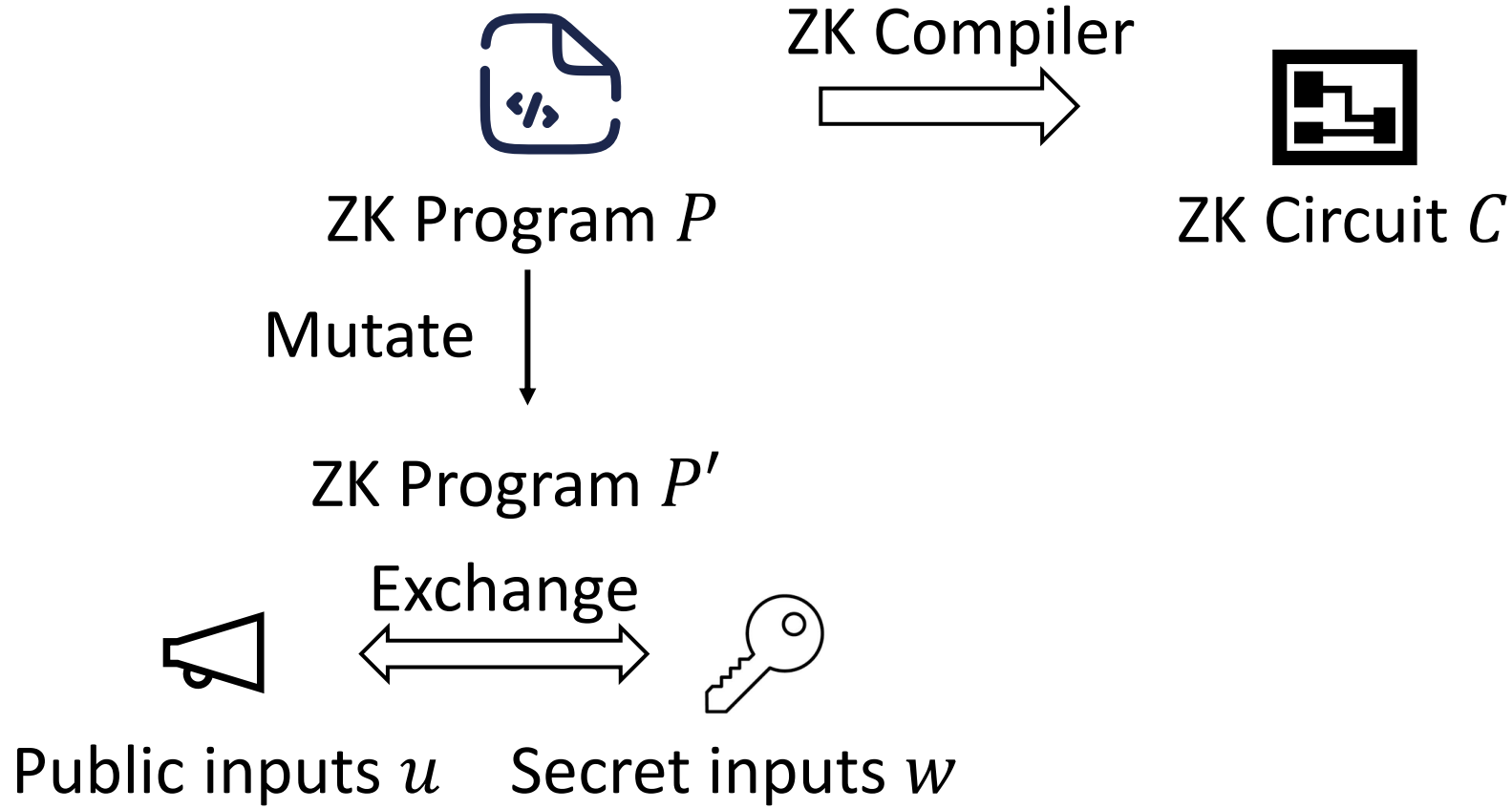


```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6        assert(x <= INT_MAX) // C-MAX
7      } else {
8        x -= 1;
9        assert(x == x * 1)  // C-TAUT
10     }
11   //       C-EQ  C-AND  C-NLT
12   assert(x == 76 && !(x < 76))
13 }
```

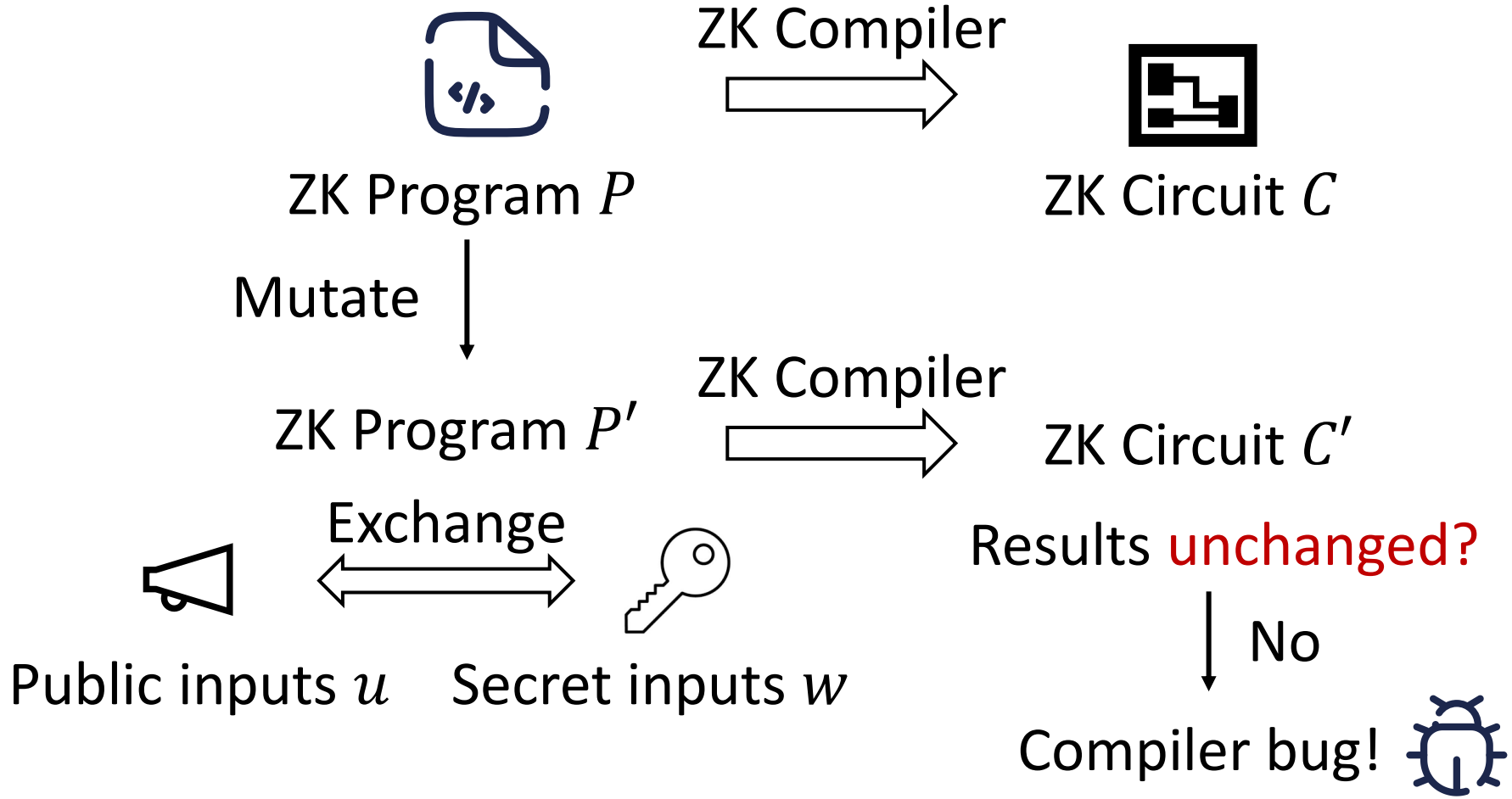
MR_{IVM}: Information Visibility Mutation



MR_{IVM}: Information Visibility Mutation

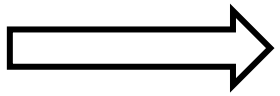


MR_{IVM}: Information Visibility Mutation



Implementation of MR_{IVM}

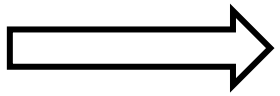
```
1 fn foo() {
2   let mut x = 81;
3   for i in 1..10
4     if x % i == 0 {
5       x += 1;
6     } else {
7       x -= 1;
8     }
9 }
```



```
1 fn foo(public u) {
2   let mut x = 81;
3   for i in 1..10
4     if x % i == u { // 0 -> u
5       x += 1;
6     } else {
7       x -= 1;
8     }
9 }
```

Implementation of MR_{IVM}

```
1  fn foo() {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == 0 {
5        x += 1;
6      } else {
7        x -= 1;
8      }
9  }
```



```
1  fn foo(public u, private w) {
2    let mut x = 81;
3    for i in 1..10
4      if x % i == u { // 0 -> u
5        x += 1;
6      } else {
7        x -= w;       // 1 -> w
8      }
9  }
```

Testing Target Selection

ZK Compilers	Blockchains
ZoKrates	polygon
Leo	Aleo
Noir	Aztec
CAIRO	STARKNET

Securing billions of dollars

Bug Findings

	ZoKrates	Noir	Cairo	Leo	Total
Total	7	7	5	2	21
#Confirmed	4	7	5	0	16
#Fixed	3	7	5	0	15

Infinite Withdrawal in ZoKrates

```
1 fn withdraw(account: &mut field, amount: field) {
2   let savings: field = read_savings(account);
3   assert(savings >= amount);
4   ... // Withdraw the money
5 }
```

Constraint always holds when `amount = MAX_FIELD`

Withdraw as much as you want!

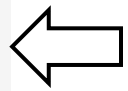
Secret Backdoors in Noir

```
1  fn low_income(pwd, salary){
2    let m = 12;
3    let avg_salary = salary / m;
4
5    // Complex data flow
6    for ...
7    // Stealthily set m to 0
8
9    // Backdoor
10   if (pwd == MAGIC)
11     avg_salary = salary / m;
12
13   assert(avg_salary < 100);
14 }
```

⇦ Division-by-zero

Secret Backdoors in Noir

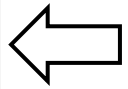
```
1  fn low_income(pwd, salary){
2    let m = 12;
3    let avg_salary = salary / m;
4
5    // Complex data flow
6    for ...
7    // Stealthily set m to 0
8
9    // Backdoor
10   if (pwd == MAGIC)
11     avg_salary = salary / m;
12
13   assert(avg_salary < 100);
14 }
```



avg_salary ← 0

Secret Backdoors in Noir

```
1  fn low_income(pwd, salary){
2    let m = 12;
3    let avg_salary = salary / m;
4
5    // Complex data flow
6    for ...
7    // Stealthily set m to 0
8
9    // Backdoor
10   if (pwd == MAGIC)
11     avg_salary = salary / m;
12
13   assert(avg_salary < 100);
14 }
```



0 < 100 Passed

Secret Backdoors in Noir

```
1  fn low_income(pwd, salary){
2    let m = 12;
3    let avg_salary = salary / m;
4
5    // Complex data flow
6    for ...
7    // Stealthily set m to 0
8
9    // Backdoor
10   if (pwd == MAGIC)
11     avg_salary = salary / m;
12
13   assert(avg_salary < 100);
14 }
```

⇦ Backdoor

Input `pwd = MAGIC` and receive benefits for low-incomers!

Summary

- The first work to uncover bugs in ZK compilers
- Approach: two mutations:
  - Satisfiability-invariant mutation
  - Information visibility mutation
- Findings: 21 bugs on four mainstream ZK compilers

Summary

- The first work to uncover bugs in ZK compilers
- Approach: two mutations:
  - Satisfiability-invariant mutation
  - Information visibility mutation
- Findings: 21 bugs on four mainstream ZK compilers

Thanks for listening!