Secure Compilation (?)

Øresund Security Days

22 May, 2019

Frédéric Besson, Alexandre Dang, and Thomas Jensen

Compilers introduce vulnerabilities!

Preservation of semantic correctness

- Lots of work have been done on semantic preservation of compilers
- CompCert, Vellvm, CakeML, etc.

Gap between correctness and security [D'Silva et al., SPW15]

- Correctness is not enough
- Does not guarantee preservation of security properties enforced at source level
- Compilation may
  - Introudce timing channels in carefuylly crafted constant-time code,
  - remove security-critical instructions.

Information-Flow Preservation

- Programs deal with sensitive data that we do not want to reveal
- Security guideline: "do not store secret data longer than necessary".
- But what would a moderately intelligent compiler do with this code?

```
crypt(key, t):
   c = key ^ t
   key = 0
   return c
```

Information-Flow Preservation

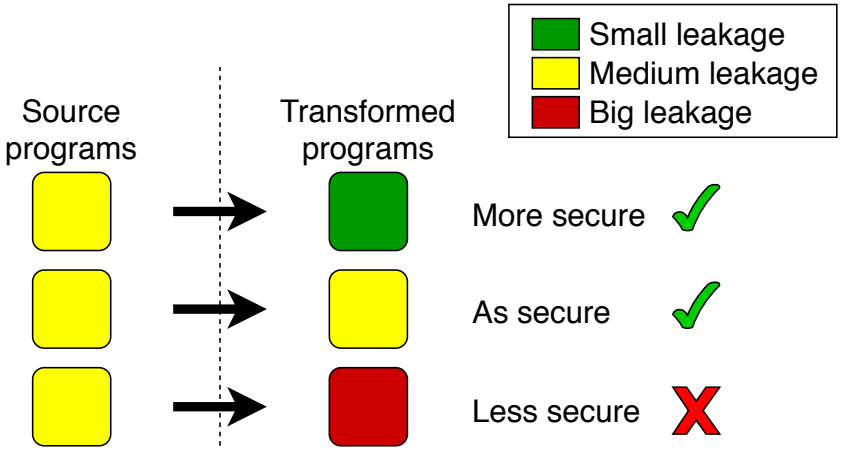
- Current programs deal with sensitive data that we do not want to leak
- Protections are not always preserved during compilation
- Simple example of *Dead Store Elimination*

```
crypt(key, t):
   c = key ^ t
   key = 0
   return c
```

```
crypt(key, t):
   c = key ^ t
   Skip
   return c
```

Preservation and not enforcement

- Our goal is not to produce secure programs
- Transformed programs should be at least as secure as their source



Our work

Objective

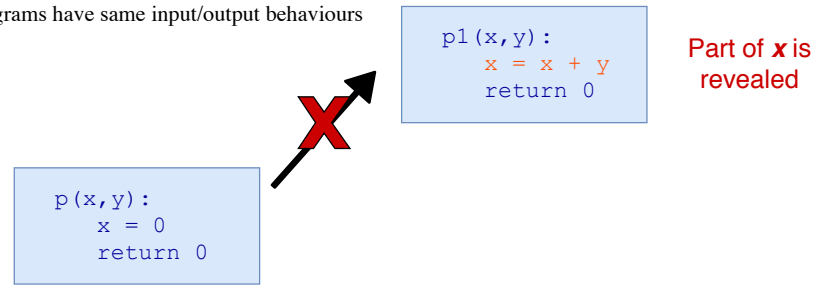
Preventing a transformed program from leaking more information
than the source program

Content of the talk

- Formal definition of an IFP* transformation
- Proof technique to certify that a transformation is IFP
- Application to two compiler transformations
  - Secure *Dead Store Elimination*
  - Validator for *Register Allocation* *Information Flow Preserving

Getting familiar with IFP

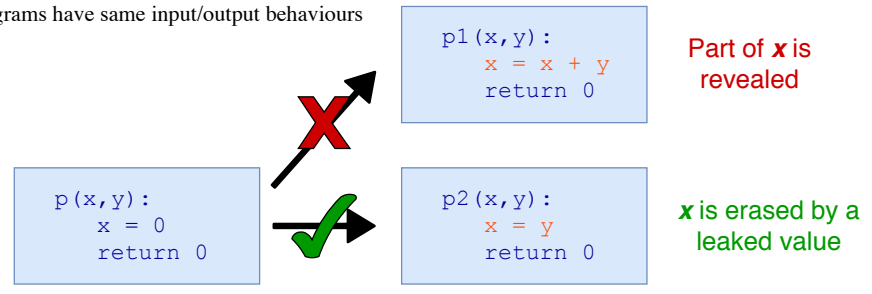
The programs have same input/output behaviours



Part of ***x*** is revealed

Getting familiar with IFP

The programs have same input/output behaviours

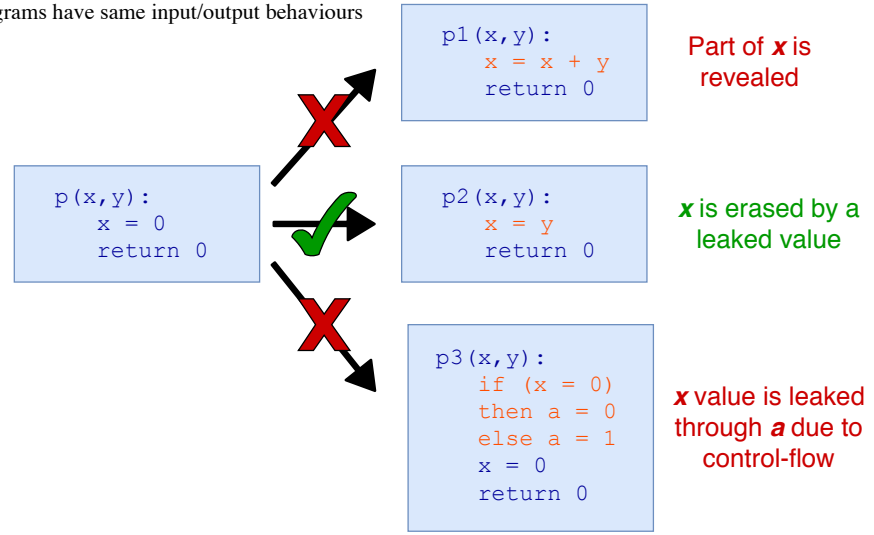


Part of ***x*** is revealed

x is erased by a leaked value

Getting familiar with IFP

The programs have same input/output behaviours

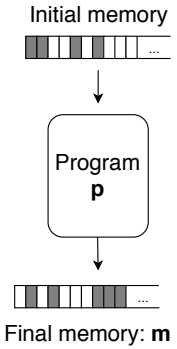


Part of ***x*** is revealed

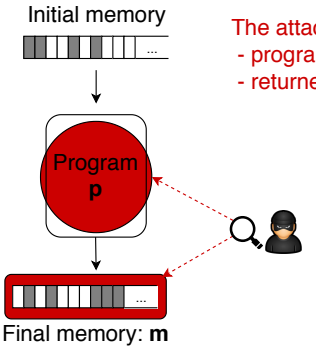
x is erased by a leaked value

x value is leaked through ***a*** due to control-flow

Execution model

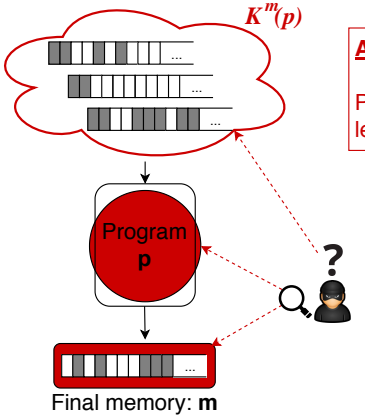


Attacker model



The attacker observes:
- program code
- returned memory

Attacker knowledge [Askarov and Sabelfeld, CSF12]



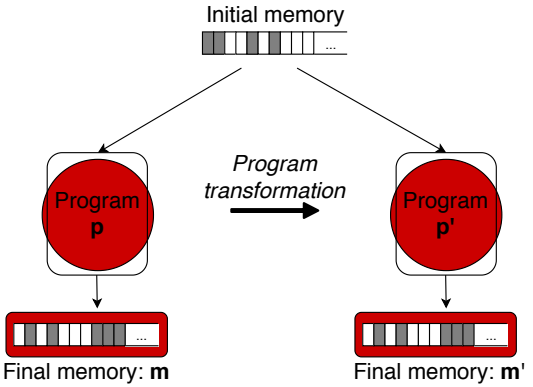
Attacker knowledge

Possible initial memories leading to the final memory ***m***

Attacker knowledge [Askarov and Sabelfeld, CSF12]

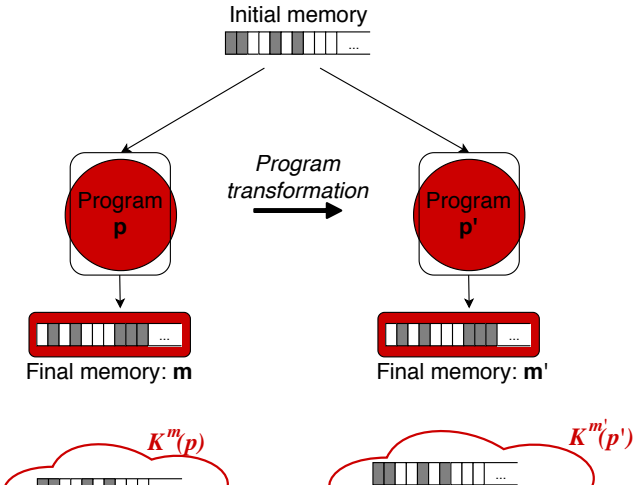
IFP transformation

p' is at least as secure as p iff:



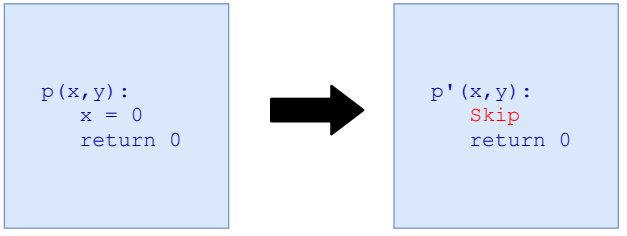
IFP transformation

p' is at least as secure as p iff:



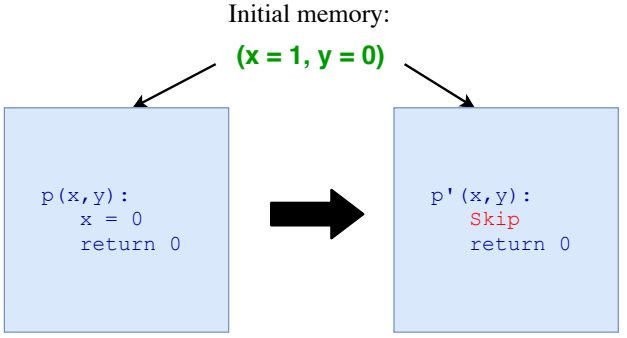
Is it the right definition? (1/2)

- We want to reject this transformation from p to p'
- We need to find a counter-example to IFP where $\mathcal{K}^m(p) \not\sim \mathcal{K}^{m'}(p')$



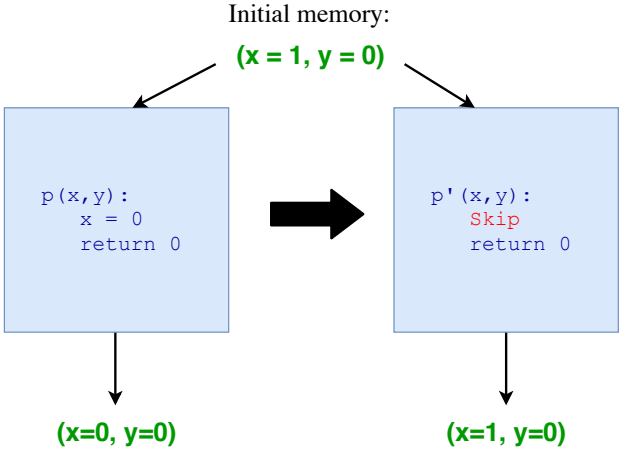
Is it the right definition? (1/2)

- We want to reject this transformation from p to p'
- We need to find a counter-example to IFP where $\kappa^m(p) \not\sim \kappa^{m'}(p')$



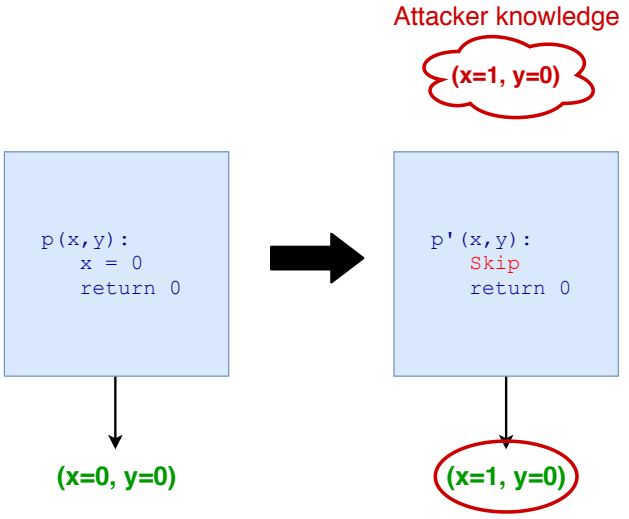
Is it the right definition? (1/2)

- We want to reject this transformation from p to p'
- We need to find a counter-example to IFP where $\kappa^m(p) \neq \kappa^{m'}(p')$



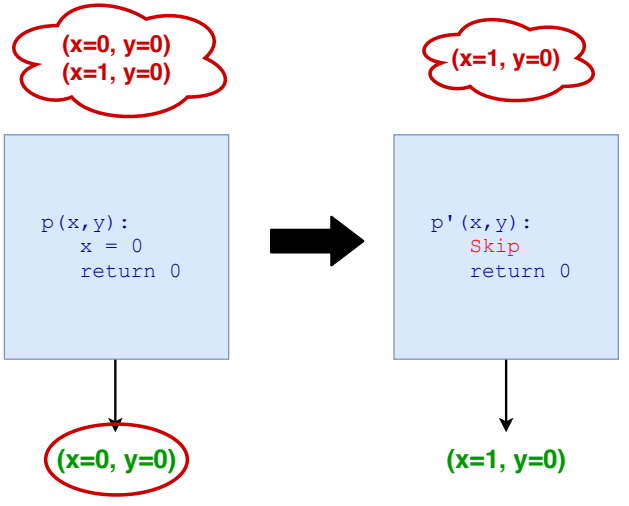
Is it the right definition? (1/2)

- We want to reject this transformation from p to p'
- We need to find a counter-example to IFP where $\mathcal{K}^m(p) \not\sim \mathcal{K}^{m'}(p')$



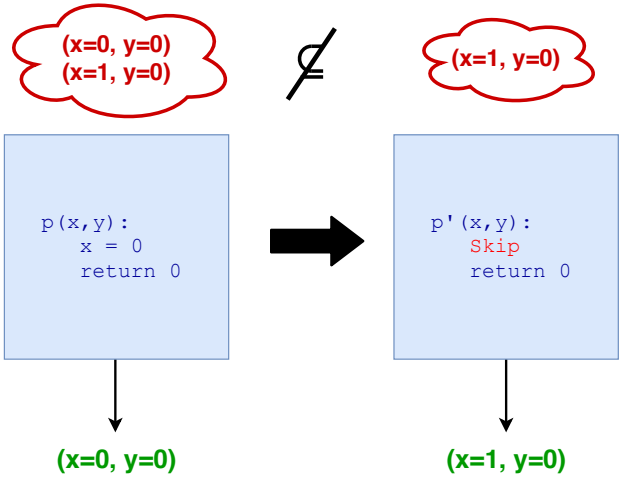
Is it the right definition? (1/2)

- We want to reject this transformation from p to p'
- We need to find a counter-example to IFP where $\mathcal{K}^m(p) \not\sim \mathcal{K}^{m'}(p')$



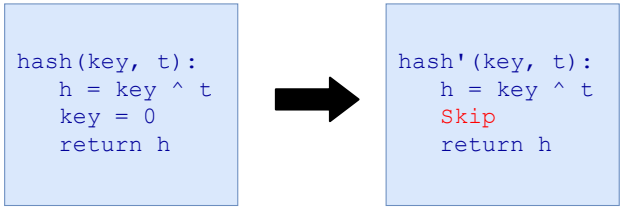
Is it the right definition? (1/2)

- We want to reject this transformation from p to p'
- We need to find a counter-example to IFP where $\mathcal{K}^m(p) \not\sim \mathcal{K}^{m'}(p')$



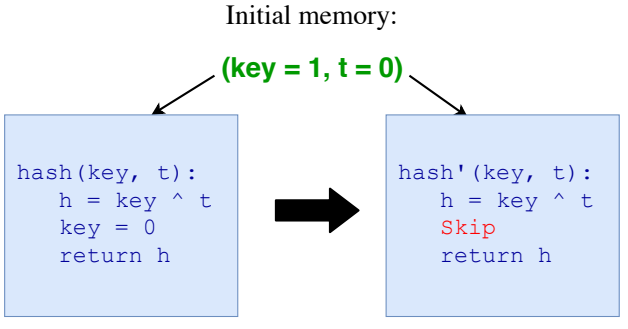
Is it the right definition? (2/2)

- We want to reject this transformation from $hash$ to $hash'$
- We need to find a counter-example to IFP where $K^m(hash) \not\subset K^{m'}(hash')$



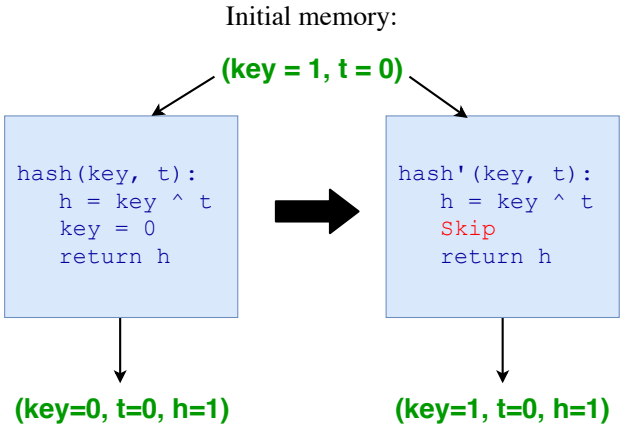
Is it the right definition? (2/2)

- We want to reject this transformation from $hash$ to $hash'$
- We need to find a counter-example to IFP where $K^m(hash) \not\subset K^{m'}(hash')$



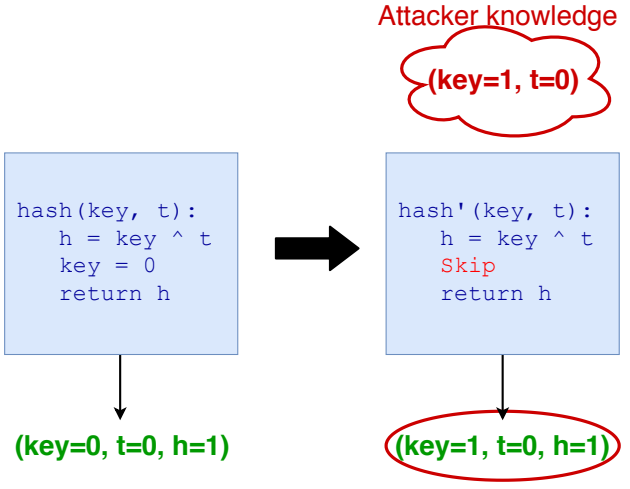
Is it the right definition? (2/2)

- We want to reject this transformation from $hash$ to $hash'$
- We need to find a counter-example to IFP where $K^m(hash) \not\subset K^{m'}(hash')$



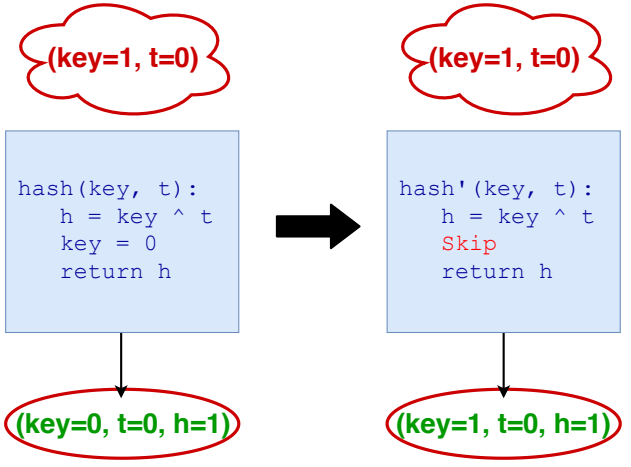
Is it the right definition? (2/2)

- We want to reject this transformation from $hash$ to $hash'$
- We need to find a counter-example to IFP where $K^m(hash) \not\subset K^{m'}(hash')$



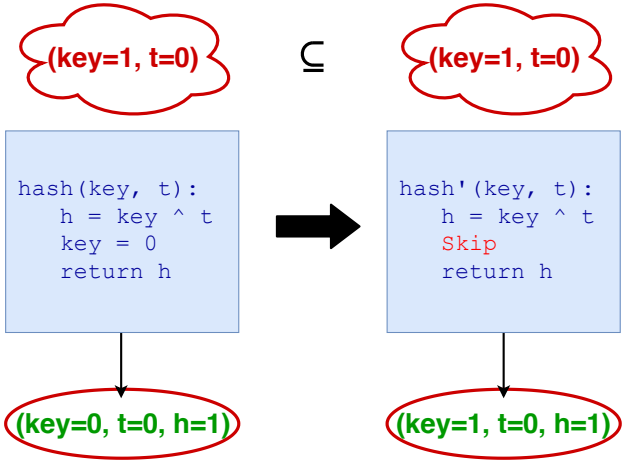
Is it the right definition? (2/2)

- We want to reject this transformation from $hash$ to $hash'$
- We need to find a counter-example to IEP where $K^m(hash) \not\subset K^{m'}(hash')$



Is it the right definition? (2/2)

- We want to reject this transformation from $hash$ to $hash'$
- We need to find a counter-example to IFP where $K^m(hash) \not\subset K^{m'}(hash')$



Choosing the right adversary

Imagine that you are appointed manager of the DTU football team ...

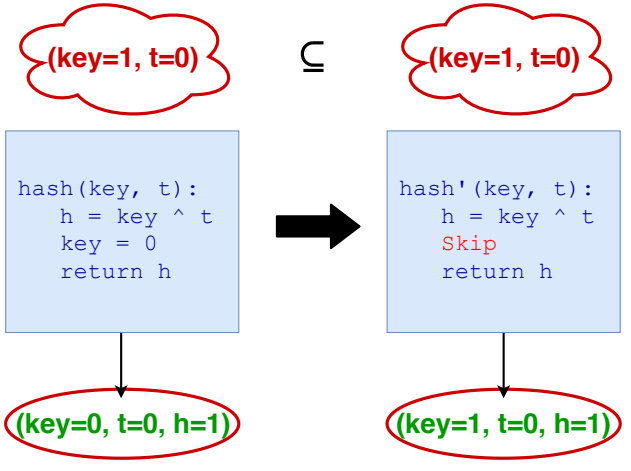
Choosing the right adversary

Imagine that you are appointed manager of the DTU football team ...

A good choice of adversary?

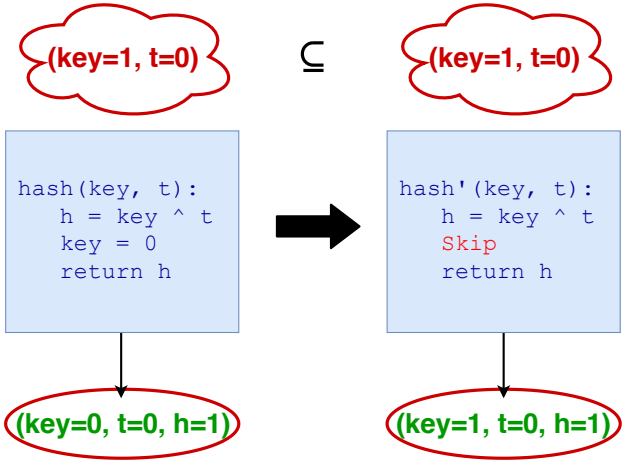
Shortcomings of the current model

- Leakage is the same in *hash'* and *hash* but is easier to obtain in *hash'*
- Our attacker is **not precise enough** and does not capture the subtelty

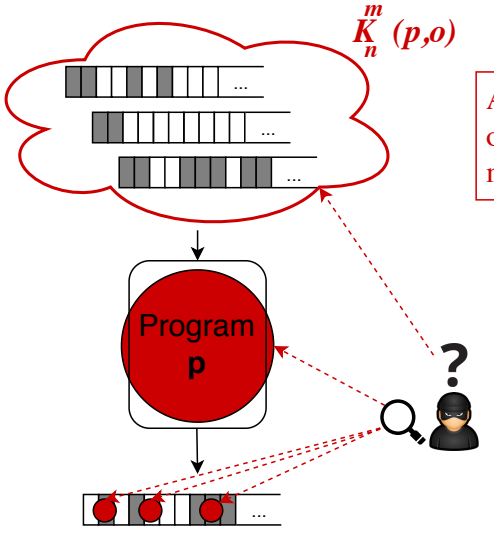


Shortcomings of the current model

- Leakage is the same in *hash'* and *hash* but is easier to obtain in *hash'*
- Our attacker is **too powerful** and does not capture the subtelty
- **We add attackers with smaller observation power**



Partial Attacker Knowlege

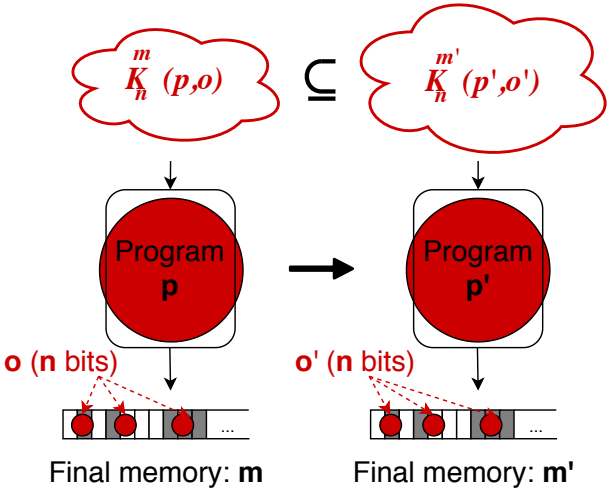


A partial attacker makes an observation **o** of **n** bits on the final memory **m**

Remark

Partial attacker knowledge with observation power $\mathbf{n} \rightarrow \infty$ is the same as attacker knowledge

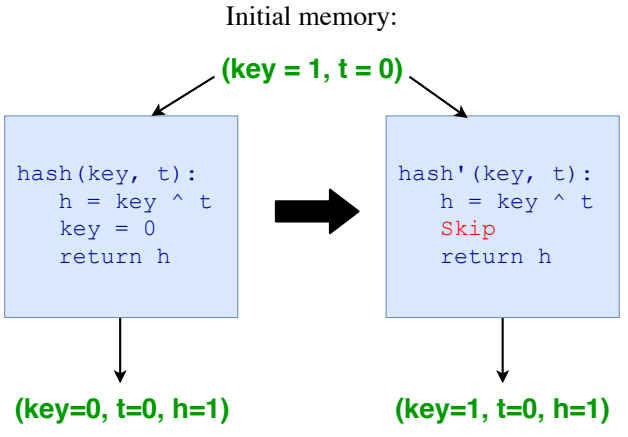
New Security Relation



Secure Relation

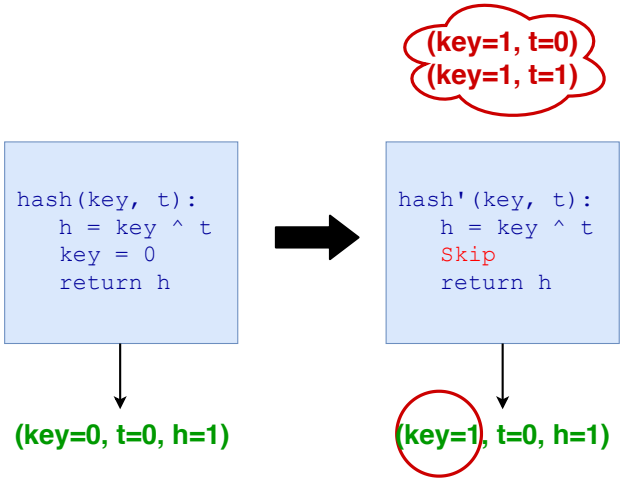
Back to our example

- We want to reject this transformation from $hash$ to $hash'$
- We need a counter-example to IFP where $\forall o.\ K_{\sim}^{m}(p, o) \not\subset K_{\sim}^{m'}(p', o')$



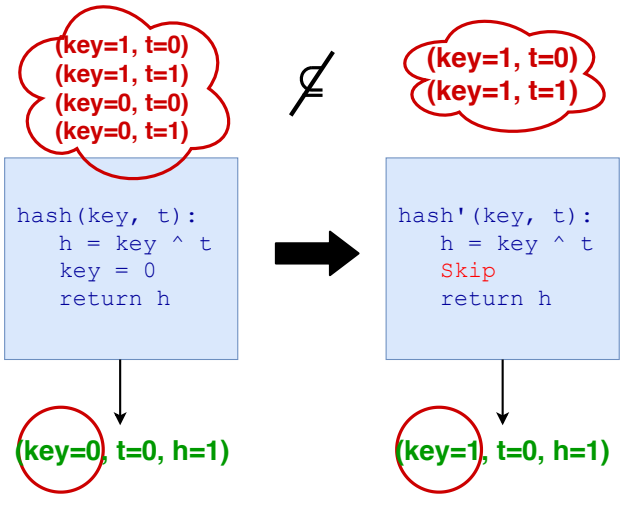
Back to our example

- We want to reject this transformation from $hash$ to $hash'$
- We need a counter-example to IFP where $\forall o, K_o^m(p, o) \not\subset K_o^{m'}(p', o')$



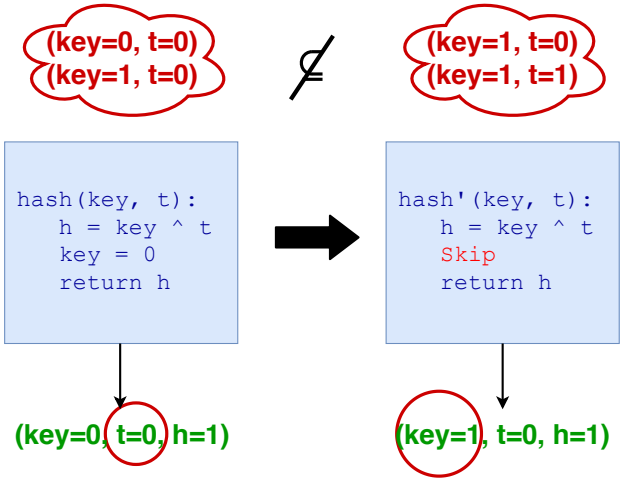
Back to our example

- We want to reject this transformation from $hash$ to $hash'$
- We need a counter-example to IFP where $\forall o, K_{\cdot}^{m}(p, o) \not\subset K_{\cdot}^{m'}(p', o')$



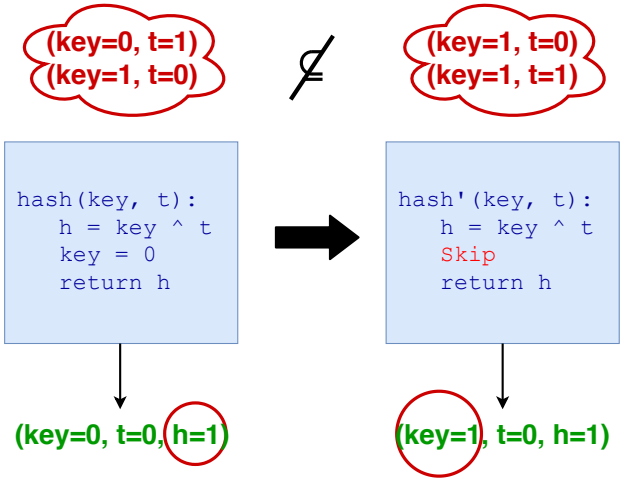
Back to our example

- We want to reject this transformation from $hash$ to $hash'$
- We need a counter-example to IFP where $\forall o, K_{\cdot}^{m}(p, o) \not\subset K_{\cdot}^{m'}(p', o')$



Back to our example

- We want to reject this transformation from $hash$ to $hash'$
- We need a counter-example to IFP where $\forall o, K_{\cdot}^{m}(p, o) \not\subset K_{\cdot}^{m'}(p', o')$



Using our IFP definition

Proof technique

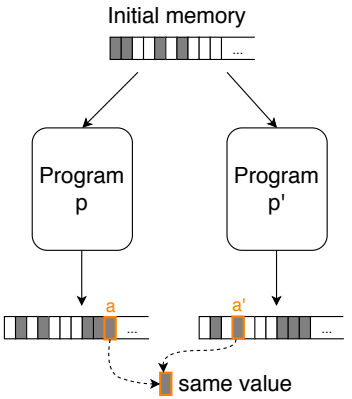
- Introduction of a proof technique to prove that a transformation is IFP

Application

Demonstration of our proof technique on two compiler transformations:

1. secure *Dead Store Elimination*
2. validator for *Register Allocation*

Proof technique

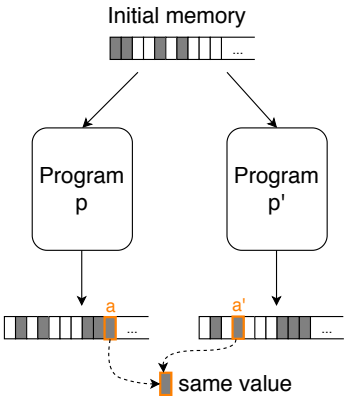


Matching addresses

Values at addresses a and a' are always equal in the final memories:

$$\_ \_ \_ \_'$$

Proof technique

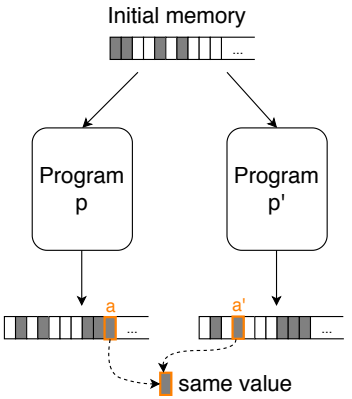


Matching addresses

Values at addresses a and a' are always equal in the final memories:

$$\_ \_ \_ \_'$$

Proof technique

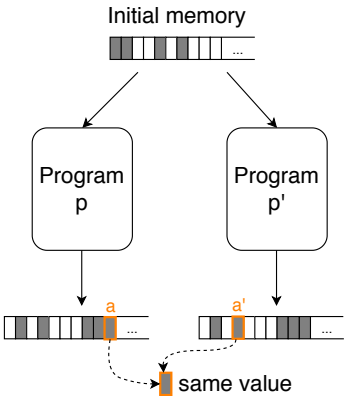


Matching addresses

Values at addresses a and a' are always equal in the final memories:

$$\_ \_ \_ \_'$$

Proof technique

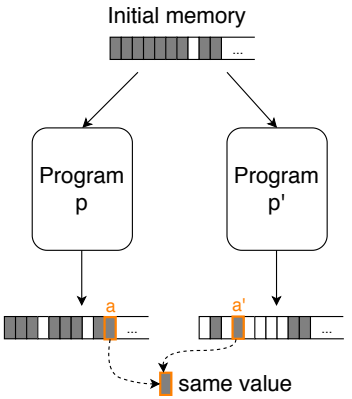


Matching addresses

Values at addresses a and a' are always equal in the final memories:

$$\_ \_ \_ \_'$$

Proof technique



Theorem

$$\forall a'.\ \exists a.\quad p \ {}_a{\equiv}_{a'}\ p'$$
$$\Rightarrow$$

Dead Store Elimination

Application of our proof technique

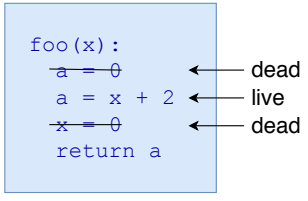
- We want to use our proof technique to create an IFP DSE*
- source and transformed programs must return identical final memories
- this is not the case for a classic DSE

*Dead Store Elimination

Classic DSE

Principle

- Remove write instructions that do not affect the **returned value**

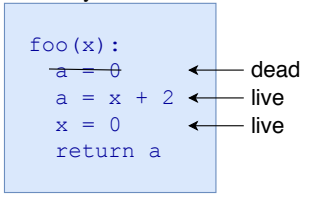


The final value of **a** is preserved

Secure DSE

Modified liveness analysis

- Source and transformed programs must return the same final memory

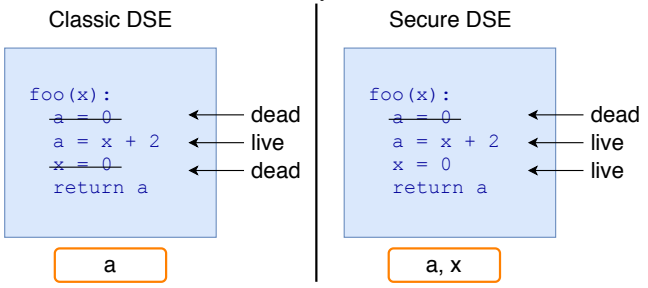


We preserve the whole final memory.
Here: **a** and **x**

- Keep "stores" that affect the **whole final memory**

Comparison

- less aggressive optimization
- preserves the state of the final memory



Register Allocation breaks IFP

A register allocation to an architecture having 2 registers

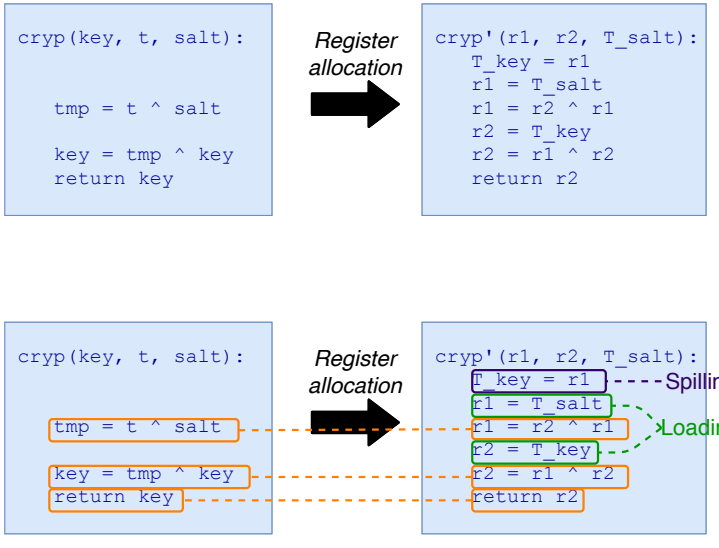
```
cryp(key, t, salt):


   tmp = t ^ salt

   key = tmp ^ key
   return key
```

Register allocation

```
cryp'(r1, r2, T_salt):
   T_key = r1
   r1 = T_salt
   r1 = r2 ^ r1
   r2 = T_key
   r2 = r1 ^ r2
   return r2
```

Register Allocation breaks IFP

A register allocation to an architecture having 2 registers



Register Allocation breaks IFP

A register allocation to an architecture having 2 registers

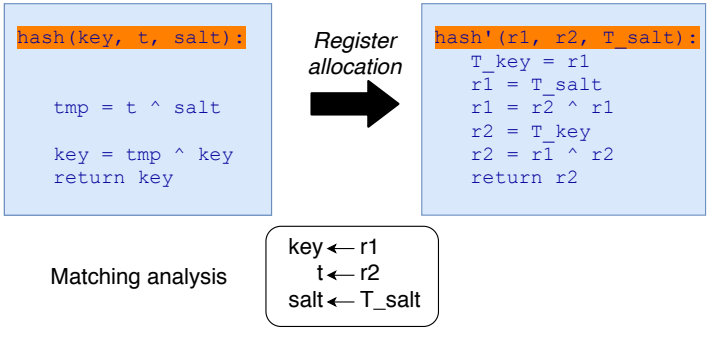
```
cryp(key, t, salt):


   tmp = t ^ salt

   key = tmp ^ key
   return key
```

Register allocation

```
cryp'(r1, r2, T_salt):
   T_key = r1
   r1 = T_salt
   r1 = r2 ^ r1
   r2 = T_key
   r2 = r1 ^ r2
   return r2
```

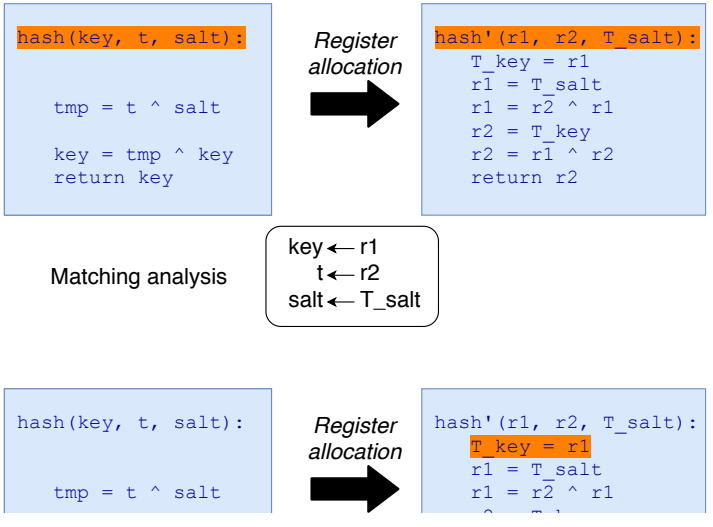
Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



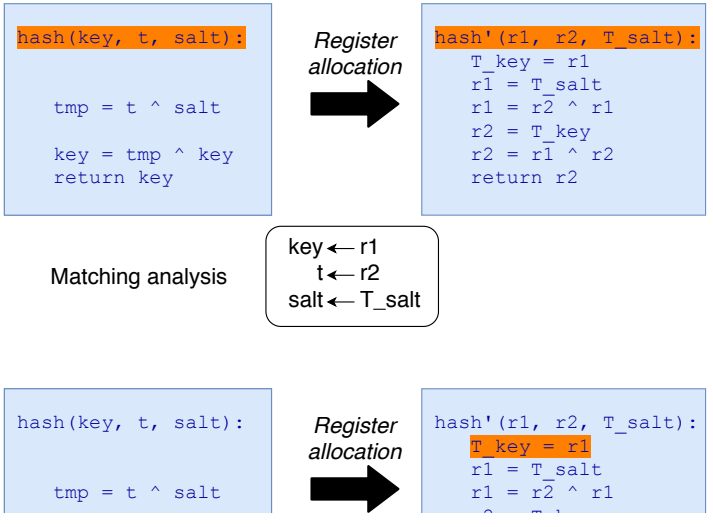
Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



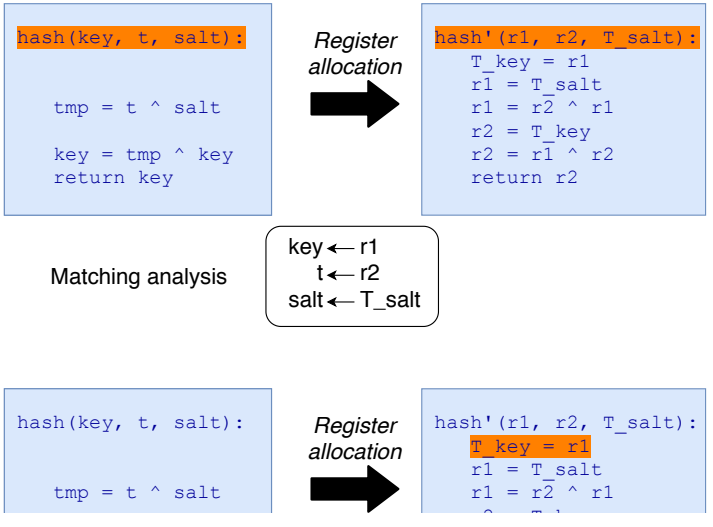
Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



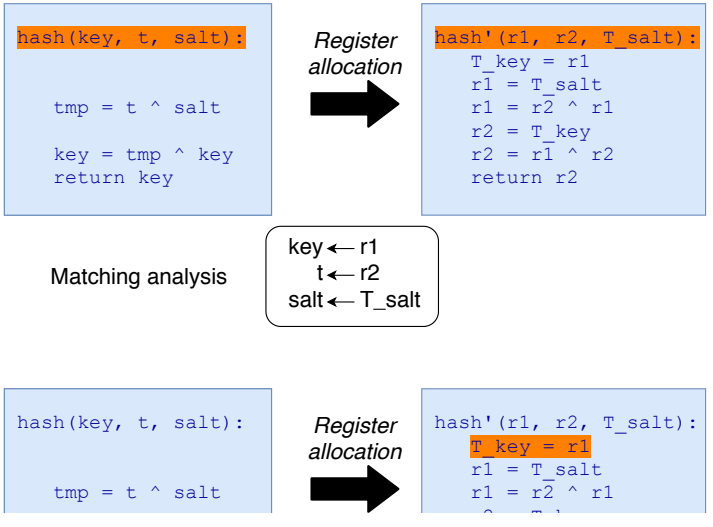
Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



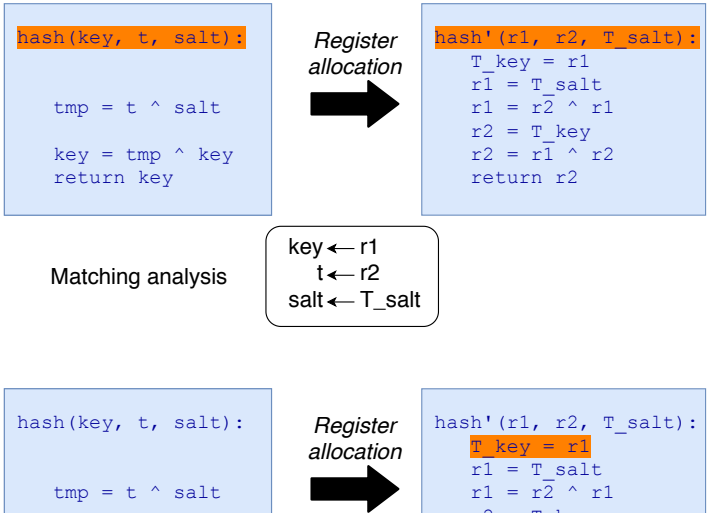
Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



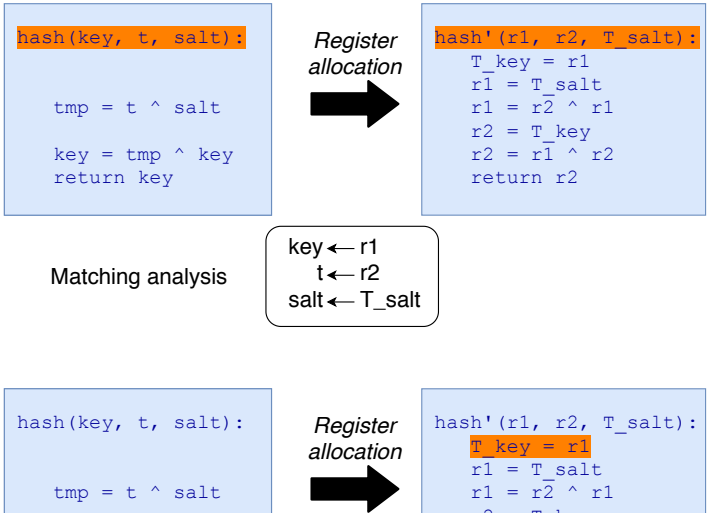
Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



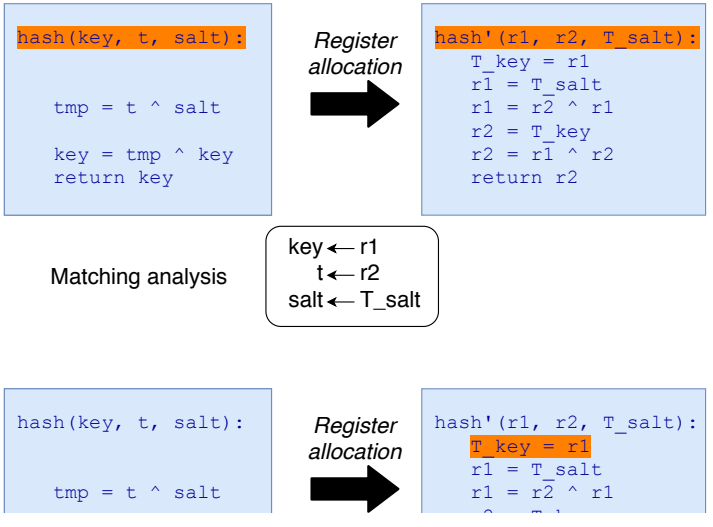
Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



Validating Register Allocation

- Validator goal: $\forall a'.\, \exists a.\quad hash\ {}_{a}{\equiv}_{a'}\ hash'$
- Build an analysis matching locations of *hash'* to locations of *hash*



Securing Register Allocation

We patch the previous leakage using our matching analysis

```
hash(key, t, salt):

    tmp = t ^ salt

    key = tmp ^ key

    return key
```

```
hash'(r1, r2, T_salt):
    T_key = r1
    r1 = T_salt
    r1 = r2 ^ r1
    r2 = T_key
    r2 = r1 ^ r2
    T_key = r1
    return r2
```

Matching analysis

tmp ← r1
key ← r2
salt ← T_salt
tmp ← T_key

Transformation is IFP

Related Work

Preservation of non-interferent programs [Deng and Namjoshi, SAS16]

- preserve programs which do not leak
- does not differentiate between leaking 1 bit or n bits

Preservation of side-channel countermeasures [Barthe et al., CSF18]

- framework to preserve security properties against side-channels
- use 2-simulation property that we can avoid

Summary

Formal definition of IFP

- Partial Attacker Knowledge able to capture multiple level of leaks
- Security relation between two programs

Applications

- Proof technique to prove that a transformation is IFP
- Secure *Dead Store Elimination*
- *Register Allocation*
  - Validation algorithm
  - Secure transformation using the validation

Other results

Stronger preservation property

- Current property is end-to-end
- Works also for stronger attacker model making multiple observations during an execution (see CSF'19).

Application to other compilation transformations

- Transformations breaking IFP
- Inlining, code motion, ...

Development

- Implemented an actual DSE and validator for CompCert
- Evaluation of cost of repairing security leaks:
  - Code size increase about 20%
  - but the instructions are rarely executed.