On the Secure Compilation of the Constant-Time Policy

Quantitative Information Flow

Luigi D. C. Soares
(luigi.domenico@dcc.ufmg.br)

Side Channels

Side Channels

Side Channels

- Outputs of a computer system

Side Channels

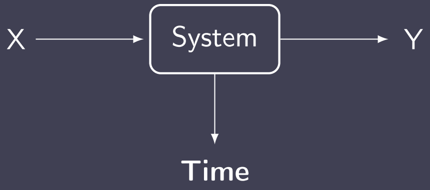
- Outputs of a computer system
- **Usually unintentional**

Side Channels

- Outputs of a computer system
- Usually unintentional

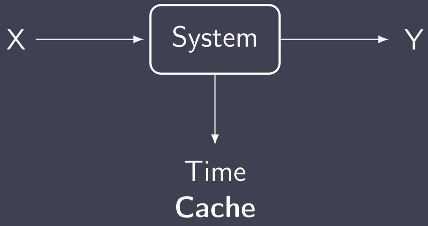
Side Channels

- Outputs of a computer system
- Usually unintentional



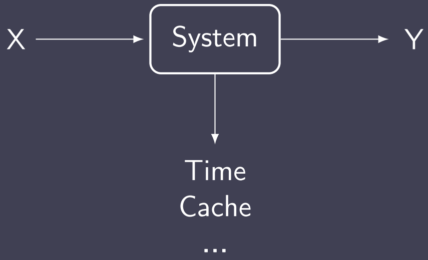
Side Channels

- Outputs of a computer system
- Usually unintentional



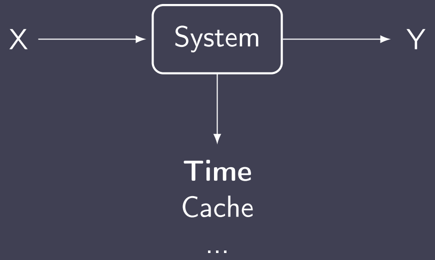
Side Channels

- Outputs of a computer system
- Usually unintentional



Side Channels

- Outputs of a computer system
- Usually unintentional



Defense Strategy

Defense Strategy

Definition 1 (Constant-Time Programming)

A program is said to implement a constant-time policy if

Defense Strategy

Definition 1 (Constant-Time Programming)

A program is said to implement a constant-time policy if its memory accesses

Defense Strategy

Definition 1 (Constant-Time Programming)

A program is said to implement a constant-time policy if its memory accesses and control flow

Defense Strategy

Definition 1 (Constant-Time Programming)

A program is said to implement a constant-time policy if its memory accesses and control flow do not depend on secret information.

Defense Strategy

Definition 1 (Constant-Time Programming)

A program is said to implement a constant-time policy if its memory accesses and control flow do not depend on secret information. In other words, the sequence of instructions and memory accesses must be the same regardless of the secret inputs.

Password Checker

Password Checker

Example 1

Consider the case of a n-bit password checker

Password Checker

Example 1

Consider the case of a n-bit password checker that either rejects the i-th bit or accepts the user's guess.

Password Checker

Example 1

Consider the case of a n-bit password checker that either rejects the i-th bit or accepts the user's guess. It could be implemented as follows:

Password Checker

Example 1

Consider the case of a n-bit password checker that either rejects the i-th bit or accepts the user's guess. It could be implemented as follows:

```
check : Vect n Bit -> Vect n Bit -> Result
check []           []           = Accept
check (One  :: g) (Zero :: pw) = Reject
check (Zero :: g) (One  :: pw) = Reject
check (_    :: g) (_    :: pw) = check g pw
```

Password Checker

Example 1

Consider the case of a n-bit password checker that either rejects the i-th bit or accepts the user's guess. It could be implemented as follows:

```
check : Vect n Bit -> Vect n Bit -> Result
check []             []              = Accept
check (One  :: g)    (Zero :: pw)    = Reject
check (Zero :: g)    (One  :: pw)    = Reject
check (_    :: g)    (_    :: pw)    = check g pw
```

Side channel

Password Checker

Example 2

Consider, again, the case of a n-bit password checker.

Password Checker

Example 2

Consider, again, the case of a n-bit password checker. But, this time it either rejects or accepts the entire guessed password.

Password Checker

Example 2

Consider, again, the case of a n-bit password checker. But, this time it either rejects or accepts the entire guessed password. It could be implemented as follows:

Password Checker

Compilers Laboratory

Example 2

Consider, again, the case of a n-bit password checker. But, this time it either rejects or accepts the entire guessed password. It could be implemented as follows:

```
check' : Vect n Bit -> Vect n Bit -> Result
check' []         []          = Accept
check' (a :: g) (b :: pw) = ctsel (r && r') Accept Reject
    where r  = check' g pw == Accept
          r' = a == b
```

Password Checker

```
check' : Vect n Bit -> Vect n Bit -> Result
check' []        []          = Accept
check' (a :: g)  (b :: pw)   = ctsel (r && r') Accept Reject
    where r  = check' g pw == Accept
          r' = a == b
```

Password Checker

```
check' : Vect n Bit -> Vect n Bit -> Result
check' []          []          = Accept
check' (a :: g) (b :: pw) = ctsel (r && r') Accept Reject
    where r  = check' g pw == Accept
          r' = a == b
```

- **Function ctsel stands for constant-time selector**

Password Checker

```
check' : Vect n Bit -> Vect n Bit -> Result
check' []       []        = Accept
check' (a :: g) (b :: pw) = ctsel (r && r') Accept Reject
    where r  = check' g pw == Accept
          r' = a == b
```

- Function ctsel stands for constant-time selector
- **Corresponds to x86's cmov**

Password Checker

```
check' : Vect n Bit -> Vect n Bit -> Result
check' []         []           = Accept
check' (a :: g) (b :: pw) = ctsel (r && r') Accept Reject
    where r  = check' g pw == Accept
          r' = a == b
```

- Function ctsel stands for constant-time selector
- Corresponds to x86's cmov
- **LLVM's x86-cmov-converter pass replaces cmovs with branches**

Password Checker

```
check' : Vect n Bit -> Vect n Bit -> Result
check' []          []            = Accept
check' (a :: g) (b :: pw) = ctsel (r && r') Accept Reject
    where r  = check' g pw == Accept
          r' = a == b
```

- Function ctsel stands for constant-time selector
- Corresponds to x86's cmov
- LLVM's x86-cmov-converter pass replaces cmovs with branches
- **How to prove that the constant-time property is preserved?**

Secure Compilation

Secure Compilation

- Barthe, Grégoire, and Laporte, “Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic “Constant-Time””

Secure Compilation

- Barthe, Grégoire, and Laporte, "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time""
- **Observational non-interference**

Secure Compilation

- Barthe, Grégoire, and Laporte, "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time""
- Observational non-interference
- **Constant-time simulations**

Secure Compilation

Secure Compilation

- A state is of the form $\{c, \rho\}$,

Secure Compilation

- A state is of the form $\{c, \rho\}$, where c is a command and ρ is an environment

Secure Compilation

- A state is of the form $\{c, \rho\}$, where c is a command and ρ is an environment
- **A program P is composed by a sequence of commands**

Secure Compilation

- A state is of the form $\{c, \rho\}$, where c is a command and ρ is an environment
- A program P is composed by a sequence of commands
- **The semantics of P is modelled by labelled transitions of the form**

$$a \xrightarrow{t}^{n} a',$$

Secure Compilation

- A state is of the form $\{c, \rho\}$, where c is a command and ρ is an environment
- A program P is composed by a sequence of commands
- **The semantics of P is modelled by labelled transitions of the form**

$$a \xrightarrow{t}^{n} a',$$

where a and a' are states, t is the leakage and n is the number of steps

Secure Compilation

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs,

Secure Compilation

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs, let $\mathcal{S}$ be the set of states,

Secure Compilation

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs, let $\mathcal{S}$ be the set of states, $\mathcal{S}_f$ the set of final states and

Secure Compilation

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs, let $\mathcal{S}$ be the set of states, $\mathcal{S}_f$ the set of final states and $\mathcal{L}$ the set of leakage.

Secure Compilation

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs, let $\mathcal{S}$ be the set of states, $\mathcal{S}_f$ the set of final states and $\mathcal{L}$ the set of leakage. Then, P is obs. non-interfering w.r.t. a binary relation ϕ on states,

Secure Compilation

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs, let $\mathcal{S}$ be the set of states, $\mathcal{S}_f$ the set of final states and $\mathcal{L}$ the set of leakage. Then, P is obs. non-interfering w.r.t. a binary relation ϕ on states, written $P \models \mathrm{ONI}(\phi)$,

Secure Compilation

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs, let $\mathcal{S}$ be the set of states, $\mathcal{S}_f$ the set of final states and $\mathcal{L}$ the set of leakage. Then, P is obs. non-interfering w.r.t. a binary relation ϕ on states, written $P \models \mathrm{ONI}(\phi)$, iff for all $a, a' \in P(\mathcal{I})$, $b, b' \in \mathcal{S}$ and $t, t' \in \mathcal{L}$,

Definition 2 (Observational Non-Interference)

Let $P(\mathcal{I})$ be the set of initial states of a program P that is given a set $\mathcal{I}$ of inputs, let $\mathcal{S}$ be the set of states, $\mathcal{S}_f$ the set of final states and $\mathcal{L}$ the set of leakage. Then, P is obs. non-interfering w.r.t. a binary relation ϕ on states, written $P \models \mathrm{ONI}(\phi)$, iff for all $a, a' \in P(\mathcal{I})$, $b, b' \in \mathcal{S}$ and $t, t' \in \mathcal{L}$,

$$a \xrightarrow{t}{}^{n} b \wedge a' \xrightarrow{t'}{}^{n} b' \wedge a \,\phi\, a' \implies t = t' \wedge (b \in \mathcal{S}_f \iff b' \in \mathcal{S}_f).$$

Secure Compilation

ompilers aboratory

Definition 3 (Lockstep Simulation)

$\approx$ is a lockstep simulation w.r.t. source and target programs S and $C = [\![S]\!]$ when

Secure Compilation

Compilers Laboratory

Definition 3 (Lockstep Simulation)

$\approx$ is a lockstep simulation w.r.t. source and target programs S and $C = [\![S]\!]$ when

1. $\forall$ source step $a \to b$ and target state α such that $a \approx \alpha$, $\exists$ a target step $\alpha \to \beta$ such that $b \approx \beta$

Secure Compilation

Definition 3 (Lockstep Simulation)

$\approx$ is a lockstep simulation w.r.t. source and target programs S and $C = [\![S]\!]$ when

1. $\forall$ source step $a \to b$ and target state α such that $a \approx \alpha$, $\exists$ a target step $\alpha \to \beta$ such that $b \approx \beta$
2. $\forall$ input parameter i, we have $S(i) \approx C(i)$

Secure Compilation

Definition 3 (Lockstep Simulation)

$\approx$ is a lockstep simulation w.r.t. source and target programs S and $C = [\![S]\!]$ when

1. $\forall$ source step $a \to b$ and target state α such that $a \approx \alpha$, $\exists$ a target step $\alpha \to \beta$ such that $b \approx \beta$
2. $\forall$ input parameter i, we have $S(i) \approx C(i)$
3. $\forall$ source and target states b and β such that $b \approx \beta$,

Secure Compilation

Definition 3 (Lockstep Simulation)

$\approx$ is a lockstep simulation w.r.t. source and target programs S and $C = [\![S]\!]$ when

1. $\forall$ source step $a \to b$ and target state α such that $a \approx \alpha$, $\exists$ a target step $\alpha \to \beta$ such that $b \approx \beta$
2. $\forall$ input parameter i, we have $S(i) \approx C(i)$
3. $\forall$ source and target states b and β such that $b \approx \beta$, we have that b is a final source state iff β is a final target state

Secure Compilation

$$\begin{array}{ccc} a & \longrightarrow & b \\ \approx \vdots & & \vdots \approx \\ \alpha & \longrightarrow & \beta \end{array}$$

Secure Compilation

Definition 4 (Lockstep CT-Simulation)

$(\equiv_S, \equiv_C)$ is a lockstep CT-simulation w.r.t. $\approx$ iff:

Secure Compilation

Definition 4 (Lockstep CT-Simulation)

$(\equiv_S, \equiv_C)$ is a lockstep CT-simulation w.r.t. $\approx$ iff:

1. $\forall$ source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$ such that $a \equiv_S a'$, and

Secure Compilation

Definition 4 (Lockstep CT-Simulation)

$(\equiv_S, \equiv_C)$ is a lockstep CT-simulation w.r.t. $\approx$ iff:

1. $\forall$ source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$ such that $a \equiv_S a'$, and $\forall$ target steps $\alpha \xrightarrow{\tau} \beta$ and $\alpha' \xrightarrow{\tau'} \beta'$ such that $a \approx \alpha$, $a' \approx \alpha'$, $\alpha \equiv_C \alpha'$, $b \approx \beta$ and $b' \approx \beta'$

Secure Compilation

Definition 4 (Lockstep CT-Simulation)

$(\equiv_S, \equiv_C)$ is a lockstep CT-simulation w.r.t. $\approx$ iff:

1. $\forall$ source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$ such that $a \equiv_S a'$, and $\forall$ target steps $\alpha \xrightarrow{\tau} \beta$ and $\alpha' \xrightarrow{\tau'} \beta'$ such that $a \approx \alpha$, $a' \approx \alpha'$, $\alpha \equiv_C \alpha'$, $b \approx \beta$ and $b' \approx \beta'$ it follows that $b \equiv_S b'$, $\beta \equiv_C \beta'$ and $\tau = \tau'$

Secure Compilation

Definition 4 (Lockstep CT-Simulation)

$(\equiv_S, \equiv_C)$ is a lockstep CT-simulation w.r.t. $\approx$ iff:

1. $\forall$ source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$ such that $a \equiv_S a'$, and $\forall$ target steps $\alpha \xrightarrow{\tau} \beta$ and $\alpha' \xrightarrow{\tau'} \beta'$ such that $a \approx \alpha$, $a' \approx \alpha'$, $\alpha \equiv_C \alpha'$, $b \approx \beta$ and $b' \approx \beta'$ it follows that $b \equiv_S b'$, $\beta \equiv_C \beta'$ and $\tau = \tau'$
2. $\forall$ pairs of input parameters i and i' such that $i \, \varphi \, i'$,

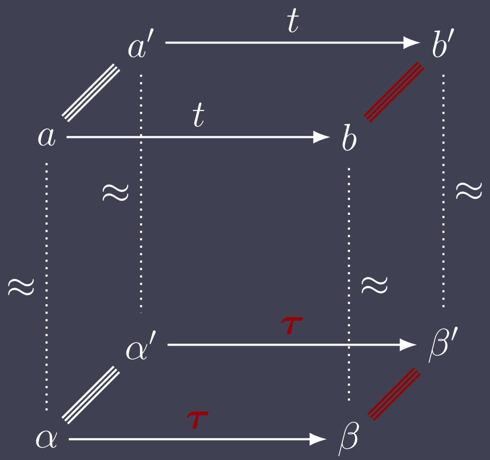
Secure Compilation

Definition 4 (Lockstep CT-Simulation)

$(\equiv_S, \equiv_C)$ is a lockstep CT-simulation w.r.t. $\approx$ iff:

1. $\forall$ source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$ such that $a \equiv_S a'$, and $\forall$ target steps $\alpha \xrightarrow{\tau} \beta$ and $\alpha' \xrightarrow{\tau'} \beta'$ such that $a \approx \alpha$, $a' \approx \alpha'$, $\alpha \equiv_C \alpha'$, $b \approx \beta$ and $b' \approx \beta'$ it follows that $b \equiv_S b'$, $\beta \equiv_C \beta'$ and $\tau = \tau'$
2. $\forall$ pairs of input parameters i and i' such that $i \, \varphi \, i'$, we have that $S(i) \equiv_S S(i')$ and $C(i) \equiv_C C(i')$, where φ is a binary relation on inputs

Secure Compilation



Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ

Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ
- Let $a.\mathrm{cmd}$ and $a.\mathrm{env}$ be the components of state a

Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ
- Let $a.\mathrm{cmd}$ and $a.\mathrm{env}$ be the components of state a

Example 3 (Constant Folding)

Constant folding reduces expressions whose operands are known. For example:

Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ
- Let $a.\mathrm{cmd}$ and $a.\mathrm{env}$ be the components of state a

Example 3 (Constant Folding)

Constant folding reduces expressions whose operands are known. For example:

1. $(\forall \rho : [e_1]_\rho = 0) \implies [\![x := e_1 * e_2]\!] = x := 0$

Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ
- Let $a.\mathrm{cmd}$ and $a.\mathrm{env}$ be the components of state a

Example 3 (Constant Folding)

Constant folding reduces expressions whose operands are known. For example:

1. $(\forall \rho : [e_1]_\rho = 0) \implies [\![x := e_1 * e_2]\!] = x := 0$
2. $(\forall \rho : [e_1]_\rho = 1) \implies [\![x := e_1 * e_2]\!] = x := e_2$

Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ
- Let $a.\mathrm{cmd}$ and $a.\mathrm{env}$ be the components of state a

Example 3 (Constant Folding)

Constant folding reduces expressions whose operands are known. For example:

1. $(\forall \rho : [e_1]_\rho = 0) \implies [\![x := e_1 * e_2]\!] = x := 0$
2. $(\forall \rho : [e_1]_\rho = 1) \implies [\![x := e_1 * e_2]\!] = x := e_2$

Thus, it suffices to define

Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ
- Let $a.\mathrm{cmd}$ and $a.\mathrm{env}$ be the components of state a

Example 3 (Constant Folding)

Constant folding reduces expressions whose operands are known. For example:

1. $(\forall\rho : [e_1]_\rho = 0) \implies [\![x := e_1 * e_2]\!] = x := 0$
2. $(\forall\rho : [e_1]_\rho = 1) \implies [\![x := e_1 * e_2]\!] = x := e_2$

Thus, it suffices to define

1. $\approx$ as $[\![a.\mathrm{cmd}]\!] = \alpha.\mathrm{cmd} \wedge a.\mathrm{env} = \alpha.\mathrm{env}$ and

Secure Compilation

- Let $[e]_\rho$ be the value of expression e under environment ρ
- Let $a.\text{cmd}$ and $a.\text{env}$ be the components of state a

Example 3 (Constant Folding)

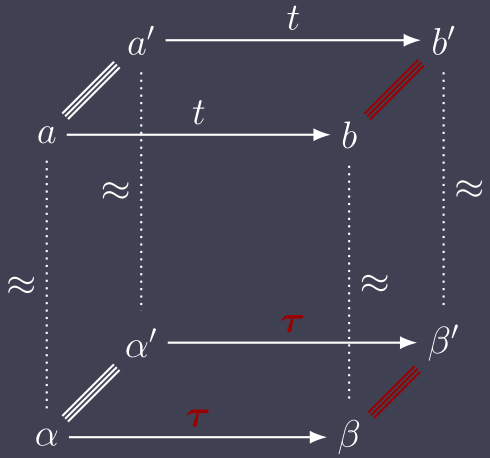
Constant folding reduces expressions whose operands are known. For example:

1. $(\forall\rho : [e_1]_\rho = 0) \implies [\![x := e_1 * e_2]\!] = x := 0$
2. $(\forall\rho : [e_1]_\rho = 1) \implies [\![x := e_1 * e_2]\!] = x := e_2$

Thus, it suffices to define

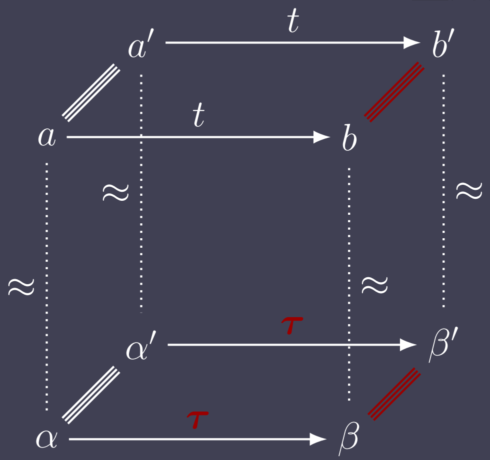
1. $\approx$ as $[\![a.\text{cmd}]\!] = \alpha.\text{cmd} \wedge a.\text{env} = \alpha.\text{env}$ and
2. $\equiv_S$ and $\equiv_C$ as $a.\text{cmd} = b.\text{cmd}$

Secure Compilation



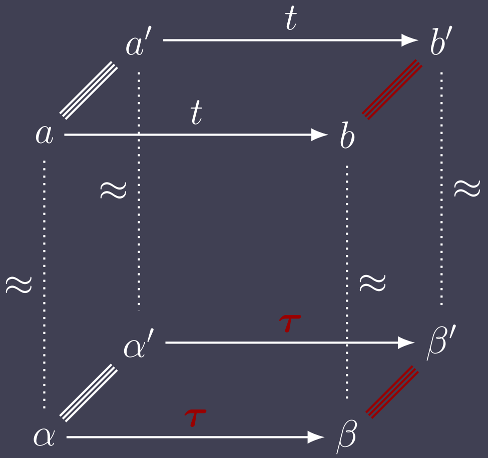
Secure Compilation

- Let $a.\mathrm{cmd}$ and $a'.\mathrm{cmd}$ be $y := A[i] * k$



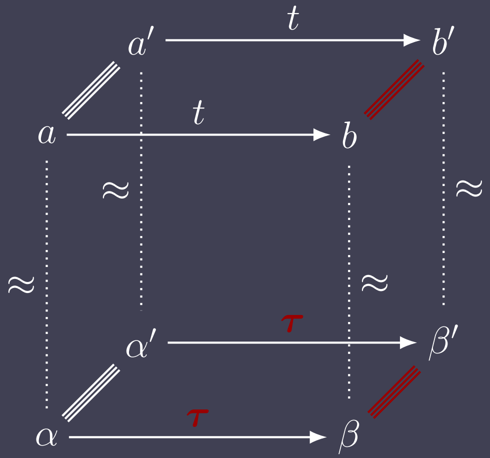
Secure Compilation

- Let a.cmd and a'.cmd be $y := A[i] * k$
- Let b.cmd and b'.cmd be $z := x + y$



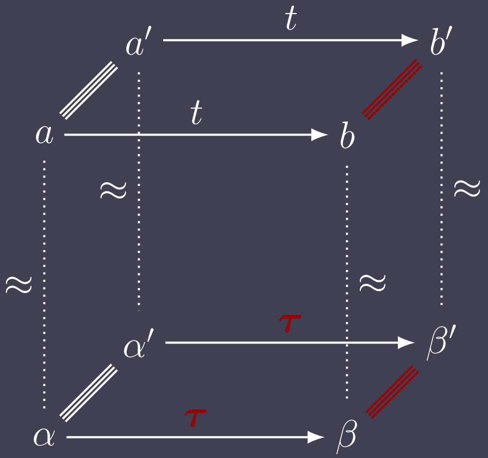
Secure Compilation

- Let $a.\mathrm{cmd}$ and $a'.\mathrm{cmd}$ be $y := A[i] * k$
- Let $b.\mathrm{cmd}$ and $b'.\mathrm{cmd}$ be $z := x + y$
- Suppose k always evaluates to 0



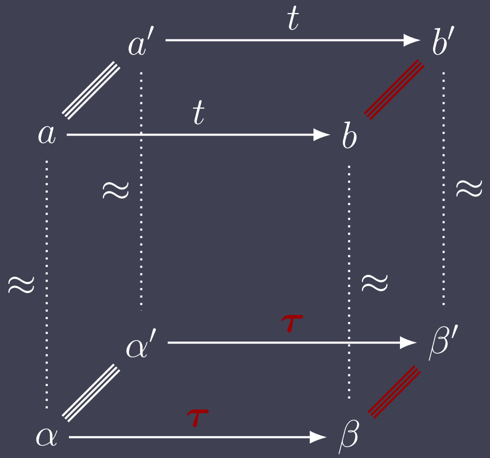
Secure Compilation

- Let $a.\mathrm{cmd}$ and $a'.\mathrm{cmd}$ be $y := A[i] * k$
- Let $b.\mathrm{cmd}$ and $b'.\mathrm{cmd}$ be $z := x + y$
- Suppose k always evaluates to 0
- Then $\alpha.\mathrm{cmd}$ $\alpha'.\mathrm{cmd}$ are $y := 0$



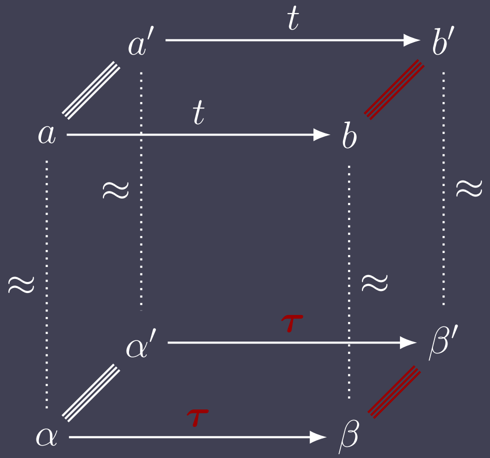
Secure Compilation

- Let a.cmd and a'.cmd be $y := A[i] * k$
- Let b.cmd and b'.cmd be $z := x + y$
- Suppose k always evaluates to 0
- Then α.cmd α'.cmd are $y := 0$
- Similarly, β.cmd and β'.cmd are $z := x$



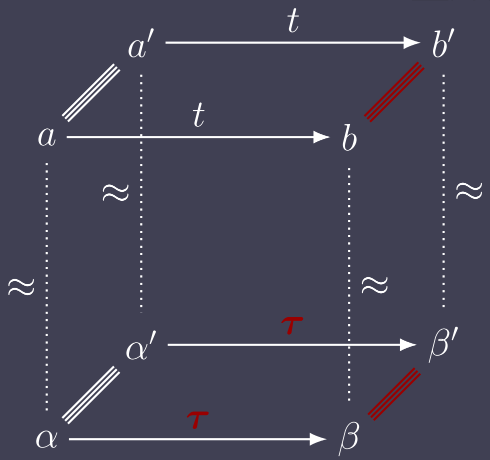
Secure Compilation

- Let a.cmd and a'.cmd be $y := A[i] * k$
- Let b.cmd and b'.cmd be $z := x + y$
- Suppose k always evaluates to 0
- Then α.cmd α'.cmd are $y := 0$
- Similarly, β.cmd and β'.cmd are $z := x$
- $t = t' \cdot (A, [i]_{\rho_a})$ and $[i]_{\rho_a} = [i]_{\rho_{a'}}$



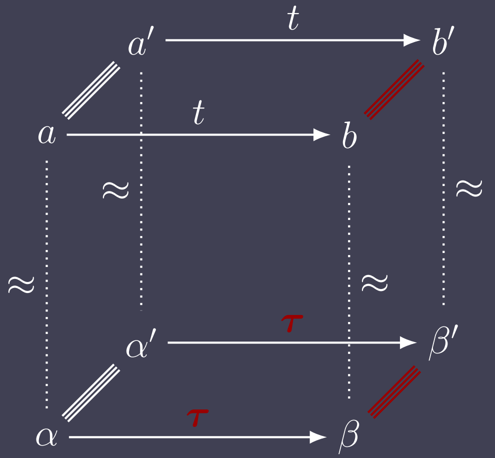
Secure Compilation

- Let a.cmd and a'.cmd be $y := A[i] * k$
- Let b.cmd and b'.cmd be $z := x + y$
- Suppose k always evaluates to 0
- Then α.cmd α'.cmd are $y := 0$
- Similarly, β.cmd and β'.cmd are $z := x$
- $t = t' \cdot (A, [i]_{\rho_a})$ and $[i]_{\rho_a} = [i]_{\rho_{a'}}$
- What is the leakage τ and is it the same in both steps?



Secure Compilation

- Let a.cmd and a'.cmd be $y := A[i] * k$
- Let b.cmd and b'.cmd be $z := x + y$
- Suppose k always evaluates to 0
- Then α.cmd α'.cmd are $y := 0$
- Similarly, β.cmd and β'.cmd are $z := x$
- $t = t' \cdot (A, [i]_{\rho_a})$ and $[i]_{\rho_a} = [i]_{\rho_{a'}}$
- What is the leakage τ and is it the same in both steps?
- $\tau = \tau'$



Relation to QIF

Relation to QIF

- Labelled transitions
- Information-theoretic channels

Relation to QIF

- Labelled transitions
- **Leakage as a trace of events**

- Information-theoretic channels
- **Leakage as a real number**

Relation to QIF

- Labelled transitions
- Leakage as a trace of events
- **Constant-time simulation**

- Information-theoretic channels
- Leakage as a real number
- **Refinement**

References

Alvim, Mário S et al. (2020). *The Science of Quantitative Information Flow*. Springer.

Barthe, Gilles, Benjamin Grégoire, and Vincent Laporte (2018). "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 328–343. DOI: `10.1109/CSF.2018.00031`.