

Mechanical Specification and Verification for Mitigating Timing-based Side Channel Leaks

Aniket Mishra, Abhishek Bichhawat

[2025-12-18 Thu]

Outline

- 1 Fall-Through Semantics for Mitigating Timing-based Side Channel Leaks
- 2 Interactive Theorem Proving - What's the Fuss About?
- 3 Mechanical Specification and Verification for Mitigating Timing-based Side Channel Leaks
- 4 Interactive Theorem Proving - An Undergraduate Perspective

An Illustrative Example

Let us consider a C program that checks some input against a given password by comparing it by character by character.

```
bool matchpwd ( int * input , size_t n ) {  
    if ( n != pwd_length ) return false;  
    for ( int i = 0; i < n ; i ++) {  
        if ( input [ i ] != pwd [ i ]) return false ;  
    }  
    return true ;  
}
```

An Exploit

Let's say the password is **101010**, an exploit may look like the following.

Example (An Attack Trace)

000000 Rejected in 1st iteration

An Exploit

Let's say the password is **101010**, an exploit may look like the following.

Example (An Attack Trace)

000000	Rejected in 1 st iteration
100000	Rejected in 3 rd iteration

An Exploit

Let's say the password is **101010**, an exploit may look like the following.

Example (An Attack Trace)

000000	Rejected in 1 st iteration
100000	Rejected in 3 rd iteration
110000	Rejected in 2 nd iteration

An Exploit

Let's say the password is **101010**, an exploit may look like the following.

Example (An Attack Trace)

000000	Rejected in 1 st iteration
100000	Rejected in 3 rd iteration
110000	Rejected in 2 nd iteration
101000	Rejected in 5 th iteration
101010	

An Exploit

Let's say the password is **101010**, an exploit may look like the following.

Example (An Attack Trace)

000000	Rejected in 1 st iteration
100000	Rejected in 3 rd iteration
110000	Rejected in 2 nd iteration
101000	Rejected in 5 th iteration
101100	Rejected in 4 th iteration

An Exploit

Let's say the password is **101010**, an exploit may look like the following.

Example (An Attack Trace)

000000	Rejected in 1 st iteration
100000	Rejected in 3 rd iteration
110000	Rejected in 2 nd iteration
101000	Rejected in 5 th iteration
101100	Rejected in 4 th iteration
101010	Password accepted!

Threat Model

Thus, given an adversary that can execute the program under security label ℓ .

Threat Model

Thus, given an adversary that can execute the program under security label ℓ .

Data Security

The adversary can view the state of the memory, but values that are high relative to ℓ are invisible.

Threat Model

Thus, given an adversary that can execute the program under security label ℓ .

Data Security

The adversary can view the state of the memory, but values that are high relative to ℓ are invisible. As an example, let us take $L \sqsubseteq M \sqsubseteq H$ and let us take $\ell = M$.

Let us say our μ is $x \rightarrow \text{true}^L, y \rightarrow \text{true}^M, z \rightarrow \text{true}^H$, then the adversary's view of the memory is:

Threat Model

Thus, given an adversary that can execute the program under security label ℓ .

Data Security

The adversary can view the state of the memory, but values that are high relative to ℓ are invisible. As an example, let us take $L \sqsubseteq M \sqsubseteq H$ and let us take $\ell = M$.

Let us say our μ is $x \rightarrow \text{true}^L, y \rightarrow \text{true}^M, z \rightarrow \text{true}^H$, then the adversary's view of the memory is:

$x \rightarrow \text{true}^L, y \rightarrow \text{true}^M, z \rightarrow *$

Threat Model

Thus, given an adversary that can execute the program under security label ℓ .

Data Security

The adversary can view the state of the memory, but values that are high relative to ℓ are invisible. As an example, let us take $L \sqsubseteq M \sqsubseteq H$ and let us take $\ell = M$.

Let us say our μ is $x \rightarrow \text{true}^L, y \rightarrow \text{true}^M, z \rightarrow \text{true}^H$, then the adversary's view of the memory is:

$x \rightarrow \text{true}^L, y \rightarrow \text{true}^M, z \rightarrow *$

Timing Security

The adversary has exact knowledge of when and where each memory access takes place during a program's execution.

Expression Semantics

CONST

$$\frac{}{n \mid \mu \Downarrow_{pc} \langle \text{CONST} \rangle, n^{pc}}$$

VAR

$$\frac{\mu(x) = n^k}{x \mid \mu \Downarrow_{pc} \langle \text{VAR}_x \rangle, n^{pc \sqcup k}}$$

OPER

$$\frac{e_1 \mid \mu \Downarrow_{pc} T_1, n_1^{k_1} \quad e_2 \mid \mu \Downarrow_{pc} T_2, n_2^{k_2} \quad n = n_1 \oplus n_2}{e_1 \oplus e_2 \mid \mu \Downarrow_{pc} \langle T_1, T_2, \text{OPER}_{\oplus} \rangle, n^{k_1 \sqcup k_2}}$$

Command Semantics

The Basics

SKIP

$$\frac{}{\text{skip} \mid \mu \Downarrow_{pc} \langle \text{SKIP} \rangle \mid \mu}$$

ASSN

$$\frac{e \mid \mu \Downarrow_{pc} T, n^k}{x := e \mid \mu \Downarrow_{pc} \langle T, \text{ASSN}_x \rangle \mid \mu, x \mapsto n^k}$$

SEQ

$$\frac{c_1 \mid \mu \Downarrow_{pc} T_1 \mid \mu' \quad c_2 \mid \mu' \Downarrow_{pc} T_2 \mid \mu''}{c_1; c_2 \mid \mu \Downarrow_{pc} \langle T_1, T_2, \text{SEQ} \rangle \mid \mu''}$$

Command Semantics

The Basics

SKIP

$$\text{skip} \mid \mu \Downarrow_{pc} \langle \text{SKIP} \rangle \mid \mu$$

ASSN

$$e \mid \mu \Downarrow_{pc} T, n^k$$
$$x := e \mid \mu \Downarrow_{pc} \langle T, \text{ASSN}_x \rangle \mid \mu, x \mapsto n^k$$

SEQ

$$\frac{c_1 \mid \mu \Downarrow_{pc} T_1 \mid \mu' \quad c_2 \mid \mu' \Downarrow_{pc} T_2 \mid \mu''}{c_1; c_2 \mid \mu \Downarrow_{pc} \langle T_1, T_2, \text{SEQ} \rangle \mid \mu''}$$

IF-HIGH

IF-HIGH

$$\frac{\text{debranch}(c_1, n, pc) \mid \mu \Downarrow_k T_1 \mid \mu' \quad e \mid \mu \Downarrow_{pc} T, n^k \quad \text{debranch}(c_2, !n, pc) \mid \mu' \Downarrow_k T_2 \mid \mu'' \quad k \not\subseteq pc}{\text{if } e \text{ then } c_1 \text{ else } c_2 \mid \mu \Downarrow_{pc} \langle T, T_1, T_2, \text{IF-HIGH} \rangle \mid \mu''}$$

Debranch Semantics: The Basics

DEB-ASSN-TRUE

$$\frac{e \mid \mu \Downarrow_{pc} T_1, n^k \quad x \mid \mu \Downarrow_{pc} T_2, n_1^{k_1}}{\text{debranch}(x := e, (\text{true}), \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \text{DEB-ASSN}_x \rangle \mid \mu, x \mapsto n^k}$$

DEB-ASSN-FALSE

$$\frac{e \mid \mu \Downarrow_{pc} T_1, n^k \quad x \mid \mu \Downarrow_{pc} T_2, n_1^{k_1}}{\text{debranch}(x := e, (\text{false}), \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \text{DEB-ASSN}_x \rangle \mid \mu, x \mapsto n_1^k}$$

Debranch Semantics: IFs

DEB-IF-HIGH

$$\frac{e_1 \mid \mu \Downarrow_{\ell} T_3, n_1^{k_{\ell}} \quad e_1 \mid \mu \Downarrow_{pc} T_4, n_1^{k_{pc}} \quad k_{\ell} \not\subseteq \ell \quad n_1 \& \& n = n' \quad !n_1 \& \& n = n''}{\text{debranch}(c_1, n', \ell) \mid \mu \Downarrow_{k_{pc}} T_1 \mid \mu' \quad \text{debranch}(c_2, n'', \ell) \mid \mu' \Downarrow_{k_{pc}} T_2 \mid \mu''}$$

$$\text{debranch}(\text{if } e_1 \text{ then } c_1 \text{ else } c_2, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, T_3, T_4, \text{DEB - IF - HIGH} \rangle \mid \mu''$$

DEB-IF-LOW

$$c = \left\{ \begin{array}{ll} \text{debranch}(c_1, n, \ell) & \text{if } n_1 = \text{true} \\ \text{debranch}(c_2, n, \ell) & \text{otherwise} \end{array} \right\} \quad \begin{matrix} e_1 \mid \mu \Downarrow_{\ell} T_1, n_1^k \\ c \mid \mu \Downarrow_{pc} T_2 \mid \mu' \quad k \sqsubseteq \ell \end{matrix}$$

$$\text{debranch}(\text{if } e_1 \text{ then } c_1 \text{ else } c_2, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \text{DEB - IF - LOW} \rangle \mid \mu'$$

Interactive Theorem Provers have been around for quite a while. However, they have been facing a lot of very *recent* adoption.

- LEAN4 (2013)
- Agda (1999)
- Coq/Rocq (1989/2025)
- Isabelle (1986)
- Automath (1967)

Expressing Mathematical Structures

```
Inductive NAT :=  
| NAT_zero  
| NAT_succ (n: NAT).  
  
Inductive EQ {A: Type} : A → A → Type :=  
| EQ_refl (x: A) : EQ x x.  
  
Inductive EXISTS {A: Type} (P: A → Type) :=  
| EXISTS_intro (witness: A) (proof: P witness) : EXISTS P.  
  
Definition All_numbers_are_reflexively_equal  
: ∀ (n: NAT), EQ n n := λ n ⇒ EQ_refl n.  
  
Theorem All_numbers_are_reflexively_equal' : ∀ (n: NAT), EQ n n.  
Proof.  
  (** Assuming some n. *)  
  intros n.  
  (** The theorem holds by definition of EQ. *)  
  constructor.  
Qed.
```

Why Bother?

The most important thing that we get from these systems is **trust**.

- **Manual verification** can be error-prone and time-consuming.
- With a theorem prover, the implementation of the core system is comparatively very **small**.
- With this base, it is much easier to **trust results** (although there are caveats we will discuss later).

Expressing Grammars

```
(** An expression is either a primitive, a variable, or a binary operation. *)
Inductive Expression :=  
| PrimitiveExpression (prim: Primitive)  
| VarExpression (x: Var)  
| BinOpExpression (binop: BinOp) (e1 e2: Expression).  
  
(** The commands in our language are:  
+ Skip, corresponding to a NO-OP  
+ Assignments  
+ Sequences (ie. perform c1 then c2)  
+ Conditionals  
+ While Loops  
*)  
Inductive Command : Type :=  
| SkipCommand  
| AssnCommand (x: Var) (e: Expression)  
| SeqCommand (c1 c2: Command)  
| IfCommand (e: Expression) (c1 c2: Command)  
| WhileCommand (e: Expression) (c: Command)
```

Expressing Semantics

```
(*** [Language: Semantics *)

(** Described here is the expression semantics. There is not much of interest to talk about here. *)
Inductive ExpressionBigStep {binop_eval: BinOp → Primitive → Primitive → Primitive} {rel: Level → Level → Type} {latticeProof: JoinSemiLattice rel}: Expression → MemStore
← Level → TimingList → Primitive → Level → Type :=
| ConstBigStep (prim: Primitive) {pc: Level} {mu: MemStore}
  : ExpressionBigStep (PrimitiveExpression prim) mu pc (SingleTiming CONST) prim pc

| VarBigStep(x: Var) {mu: MemStore} {pc j: Level} (joinProof: Join rel pc (snd (mu x)) j)
  : ExpressionBigStep (VarExpression x) mu pc (SingleTiming (VAR x)) (fst (mu x)) j

| OperBigStep {oper: BinOp} {mu: MemStore} {e1 e2: Expression} {pc k1 k2 joinK1K2: Level} {T1 T2: TimingList} {n1 n2: Primitive}
  (p1: ExpressionBigStep e1 mu pc T1 n1 k1)
  (p2: ExpressionBigStep e2 mu pc T2 n2 k2)
  (joinProof: Join rel k1 k2 joinK1K2)
  : ExpressionBigStep (BinOpExpression oper e1 e2) mu pc (T1 , T2 , (SingleTiming (OPER oper))) (binop_eval oper n1 n2) joinK1K2.
```

Expressing Invariants

```
Lemma ExpressionTimSec {binop_eval: BinOp -> Primitive -> Primitive -> Primitive} {rel: Level -> Level -> Type} {latticeProof: JoinSemiLattice rel}:
  ∀ {e: Expression} {pc1 pc2 k1 k2: Level} {mu1 mu2: MemStore} {n1 n2: Primitive} {T1 T2: TimingList},
    @ExpressionBigStep binop_eval rel latticeProof e mu1 pc1 T1 n1 k1 →
    @ExpressionBigStep binop_eval rel latticeProof e mu2 pc2 T2 n2 k2 →
      (T1 = T2).

Proof.
  intros. dependent induction e; dependent destruction X0; dependent destruction X.
   $\vdash$  reflexivity.
   $\vdash$  reflexivity.
   $\vdash$  specialize (IHe1  $\cdots$  X1 X0_1).
  specialize (IHe2  $\cdots$  X2 X0_2).
  rewrite → IHe1.
  rewrite → IHe2.
  reflexivity.
Qed.
```

Why Did I Bother?

- I started working on this project during the first semester of my second year. I was taking Discrete Math **at the same time!**

Why Did I Bother?

- I started working on this project during the first semester of my second year. I was taking Discrete Math **at the same time!**
- During December, I spent some time in IIT Delhi with Prof Vaishnavi Sundararanjan and started properly working with theorem provers.

Why Did I Bother?

- I started working on this project during the first semester of my second year. I was taking Discrete Math **at the same time!**
- During December, I spent some time in IIT Delhi with Prof Vaishnavi Sundararanjan and started properly working with theorem provers.
- After this, I got extremely anxious about my **proofs and theorems being incorrect**.

Learning Rocq

- Well documented and maintained by a team of highly qualified researchers/engineers.

Learning to write Rocq vs Learning to write Proofs

Learning Rocq

- Well documented and maintained by a team of highly qualified researchers/engineers.
- Wonderful widely available resources for learning (Software Foundations).

Learning to write Rocq vs Learning to write Proofs

Learning Rocq

- Well documented and maintained by a team of highly qualified researchers/engineers.
- Wonderful widely available resources for learning (Software Foundations).
- At the end of the day, it is a programming language! Possibility of transfer.

Learning to write Rocq vs Learning to write Proofs

Learning Rocq

- Well documented and maintained by a team of highly qualified researchers/engineers.
- Wonderful widely available resources for learning (Software Foundations).
- At the end of the day, it is a programming language! Possibility of transfer.

Learning Proofs

- Universally accepted resources do not exist (as far as I know).

Learning to write Rocq vs Learning to write Proofs

Learning Rocq

- Well documented and maintained by a team of highly qualified researchers/engineers.
- Wonderful widely available resources for learning (Software Foundations).
- At the end of the day, it is a programming language! Possibility of transfer.

Learning Proofs

- Universally accepted resources do not exist (as far as I know).
- Discrete Math courses often focus on already existing/well-studied structures.

Learning to write Rocq vs Learning to write Proofs

Learning Rocq

- Well documented and maintained by a team of highly qualified researchers/engineers.
- Wonderful widely available resources for learning (Software Foundations).
- At the end of the day, it is a programming language! Possibility of transfer.

Learning Proofs

- Universally accepted resources do not exist (as far as I know).
- Discrete Math courses often focus on already existing/well-studied structures. But (from what I know), not as much work in defining and using your own.

Learning to write Rocq vs Learning to write Proofs

Learning Rocq

- Well documented and maintained by a team of highly qualified researchers/engineers.
- Wonderful widely available resources for learning (Software Foundations).
- At the end of the day, it is a programming language! Possibility of transfer.

Learning Proofs

- Universally accepted resources do not exist (as far as I know).
- Discrete Math courses often focus on already existing/well-studied structures. But (from what I know), not as much work in defining and using your own.
- Learning by example is difficult: "The proof is trivial", or "The proof follows simply from induction on x ".

Intuition

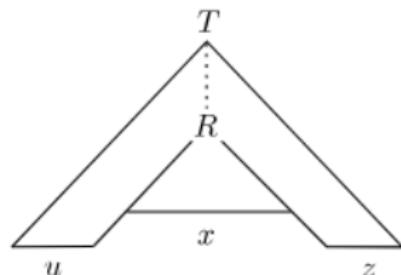
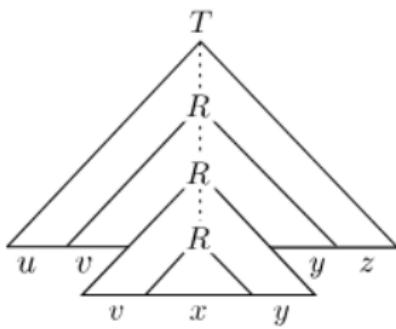
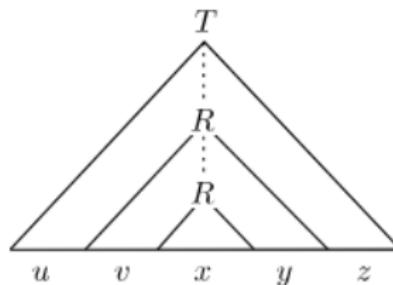
Intuition plays a big role in proofs.

Intuition

Intuition plays a big role in proofs. Take for example the pumping lemma for context free grammars.

Intuition

Intuition plays a big role in proofs. Take for example the pumping lemma for context free grammars.



Intuition in ITPs

- There is **almost no room** for intuition in ITPs.

Intuition in ITPs

- There is **almost no room** for intuition in ITPs.
- This is bad. Spending time on things obvious by intuition, may hamper the **non-obvious** things.

Intuition in ITPs

- There is **almost no room** for intuition in ITPs.
- This is bad. Spending time on things obvious by intuition, may hamper the **non-obvious** things.
- This is good. It allows for stronger guarantees.

Intuition in ITPs

- There is **almost no room** for intuition in ITPs.
- This is bad. Spending time on things obvious by intuition, may hamper the **non-obvious** things.
- This is good. It allows for stronger guarantees. **As a student, by default, your intuition is non-existent or bad.**

Costs and Caveats

Costs and Caveats

- Error in specification.

Costs and Caveats

- Error in specification.
- Bugs in prover-software. Possibility of breaking updates.

Costs and Caveats

- Error in specification.
- Bugs in prover-software. Possibility of breaking updates.
- Benefits from abstractions.

Costs and Caveats

- Error in specification.
- Bugs in prover-software. Possibility of breaking updates.
- Benefits from abstractions.
- Strictness can cause difficulty in iteration.