Fall-Through Semantics for Mitigating Timing-Based Side Channel Leaks

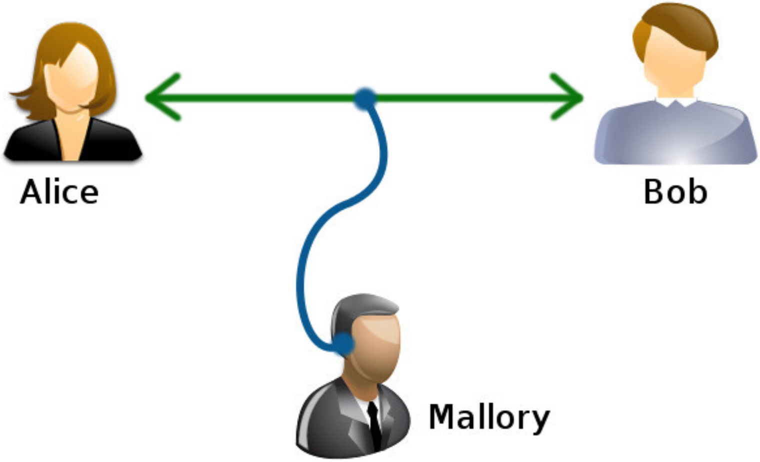
Aniket Mishra, Abhishek Bichhawat

[2025-12-18 Thu]

Outline

Traditional Cybersecurity

Traditionally, cybersecurity has dealt with the analysis of overt channels.



What are Side Channels?

A *side* channel, on the other hand, involves no such direct access. Instead, vulnerabilities are introduced by side effects of a program's execution.

What are Side Channels?

A *side* channel, on the other hand, involves no such direct access. Instead, vulnerabilities are introduced by side effects of a program's execution.

- Power used during the program's execution

What are Side Channels?

A *side* channel, on the other hand, involves no such direct access. Instead, vulnerabilities are introduced by side effects of a program's execution.

- Power used during the program's execution
- Sound generated by the machine running the program

What are Side Channels?

A *side* channel, on the other hand, involves no such direct access. Instead, vulnerabilities are introduced by side effects of a program's execution.

- Power used during the program's execution
- Sound generated by the machine running the program
- Time it takes for the program to execute

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 1^{st} 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 1st iteration
100000 .... Rejected in 3rd iteration
110000 .... Rejected in 2nd 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

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!

The Reality of Timing Side Channels

Speculative execution and cache side channels are subtler ways that these vulnerabilities can be introduced.

Dummy Code Insertion

```
volvoValue := 0;
i := 1;
while (i<=DBsize) {
  let share := sharesDB[i].name  in
  let value := lookupVal(share)*sharesDB[i].no in
    if (isVolvoShare(share))
      volvoValue := volvoValue+value
    else
      skipAsn volvoValue (volvoValue+value);
    i := i + 1
}
```

Figure 2: A padded, secure version of the program.

Constant Time Programming

Constant Time Program

```
bool equals(byte a[], size_t a_len, byte b[], size_t b_len) {
  volatile size_t x = a_len ^ b_len;
  for (size_t i = 0; ((i < a_len) & (i < b_len)); i++) {
    x |= a[i] ^ b[i];
  }
  return (x==0);
}
```

Preservation of Constant Time Property

Separately, work has been done on ensuring the preservation of this invariant. See: Formal verification of a constant-time preserving C compiler by Barthe et. al.

Code Generation

Program Repair

Work has been done in linearising the source code of a program.
See: Eliminating Timing Side-Channel Leaks using Program Repair by Wu et al.

Transforming IR

Similarly, work has also been done at generating constant time code given a non constant-time program.
See: Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization by Borrello et al.

Our Approach

- All of the above approaches require the entire source code to be available beforehand. This is not always possible.

Our Approach

- All of the above approaches require the entire source code to be available beforehand. This is not always possible.
- Many of these approaches employ the use of testing and fuzzing-based techniques to find sensitive execution pathways in the program.

Our Approach

- All of the above approaches require the entire source code to be available beforehand. This is not always possible.
- Many of these approaches employ the use of testing and fuzzing-based techniques to find sensitive execution pathways in the program.
- Thus, we propose a runtime semantics that takes care by expanding on techniques used in dynamic Information Flow Control.

Our Approach

- All of the above approaches require the entire source code to be available beforehand. This is not always possible.
- Many of these approaches employ the use of testing and fuzzing-based techniques to find sensitive execution pathways in the program.
- Thus, we propose a runtime semantics that takes care by expanding on techniques used in dynamic Information Flow Control. Furthermore, we prove that our semantics has the desired properties within the Rocq theorem prover.

The WHILE Language

$$\mu := \cdot \mid \mu, x \Rightarrow n^k$$

The WHILE Language

$$\mu := \cdot \mid \mu, x \Rightarrow n^k$$
$$e := n \mid x \mid e_1 \oplus e_2$$

The WHILE Language

$\mu := \cdot \mid \mu, x \Rightarrow n^k$
$e := n \mid x \mid e_1 \oplus \; e_2$
$c := \text{skip} \mid x := e \mid c_1; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c$

Judgements

Expression Judgement

$$e \mid \mu \Downarrow_{pc} T, n^k$$

An expression given some store μ and some evaluation level pc, evaluates to some primitive value n along with some label k, within time T.

Judgements

Expression Judgement

$$e \mid \mu \Downarrow_{pc} T, n^k$$

An expression given some store μ and some evaluation level pc, evaluates to some primitive value n along with some label k, within time T.

Command Judgement(s)

$$c \mid \mu \Downarrow_{pc} T \mid \mu' \qquad \text{debranch}(c, n, \ell) \mid \mu \Downarrow_{pc} T \mid \mu'$$

A command given some store mu, and evaluation level pc produces a store μ' in time T.

Dynamic IFC

Let us imagine there are variables x and y, where x is hidden and holds some sensitive data and y does not. There are 3 kinds of leaks that could happen.

Dynamic IFC

Let us imagine there are variables x and y, where x is hidden and holds some sensitive data and y does not. There are 3 kinds of leaks that could happen.

Example (Direct Flow)

$y := x$

Dynamic IFC

Let us imagine there are variables x and y, where x is hidden and holds some sensitive data and y does not. There are 3 kinds of leaks that could happen.

Example (Direct Flow)

$y := x$

Example (Indirect Flow)

if x then $y := 0$ else $y := 1$

Dynamic IFC

Let us imagine there are variables x and y, where x is hidden and holds some sensitive data and y does not. There are 3 kinds of leaks that could happen.

Example (Direct Flow)

$y := x$

Example (Indirect Flow)

if x then $y := 0$ else $y := 1$

Example (Side Channel)

if x then $y := 0$ else $\{y := 1;\ y := 1\}$

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 $x \to true^{L}$, $y \to true^{M}$, $z \to true^{H}$, 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 $x \rightarrow true^L$, $y \rightarrow true^M$, $z \rightarrow true^H$, then the adversary's view of the memory is:

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

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 $x \rightarrow true^L$, $y \rightarrow true^M$, $z \rightarrow true^H$, then the adversary's view of the memory is:
$x \rightarrow true^L$, $y \rightarrow true^M$, $z \rightarrow *$

Timing Security

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

Expression Semantics

CONST

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

VAR

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

OPER

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

Command Semantics

The Basics

SKIP

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

ASSN

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

SEQ

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

Command Semantics

The Basics

SKIP

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

ASSN

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

SEQ

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

IF-HIGH

IF-HIGH

$$\frac{\textsf{debranch}(c_1, n, pc) \mid \mu \Downarrow_k T_1 \mid \mu' \qquad \begin{array}{c} e \mid \mu \Downarrow_{pc} T, n^k \\ \textsf{debranch}(c_2, !n, pc) \mid \mu' \Downarrow_k T_2 \mid \mu'' \end{array} \qquad k \not\sqsubseteq pc}{\textsf{if } e \textsf{ then } c_1 \textsf{ else } c_2 \mid \mu \Downarrow_{pc} \langle T, T_1, T_2, \texttt{IF}-\texttt{HIGH} \rangle \mid \mu''}$$

Debranch Semantics: The Basics

DEB-ASSN-TRUE

$$\frac{e \mid \mu \Downarrow_{pc} T_1, n^k \qquad x \mid \mu \Downarrow_{pc} T_2, n_1^{k_1}}{\mathsf{debranch}(x := e, (true), \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \mathtt{DEB-ASSN_x} \rangle \mid \mu, x \mapsto n^k}$$

DEB-ASSN-FALSE

$$\frac{e \mid \mu \Downarrow_{pc} T_1, n^k \qquad x \mid \mu \Downarrow_{pc} T_2, n_1^{k_1}}{\mathsf{debranch}(x := e, (false), \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \mathtt{DEB-ASSN_x} \rangle \mid \mu, x \mapsto n_1^k}$$

Debranch Semantics: IFs

DEB-IF-HIGH

$$\frac{\begin{array}{c} e_1 \mid \mu \Downarrow_\ell T_3, n_1^{k_\ell} \qquad e_1 \mid \mu \Downarrow_{pc} T_4, n_1^{k_{pc}} \qquad k_\ell \not\sqsubseteq \ell \qquad n_1 \&\& n = n' \qquad !n_1 \&\& n = n'' \\ \textsf{debranch}(c_1, n', \ell) \mid \mu \Downarrow_{k_{pc}} T_1 \mid \mu' \qquad \textsf{debranch}(c_2, n'', \ell) \mid \mu' \Downarrow_{k_{pc}} T_2 \mid \mu'' \end{array}}{\textsf{debranch}(\textsf{ if } e_1 \textsf{ then } c_1 \textsf{ else } c_2, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, T_3, T_4, \mathtt{DEB-IF-HIGH} \rangle \mid \mu''}$$

DEB-IF-LOW

$$\frac{\begin{array}{c} e_1 \mid \mu \Downarrow_\ell T_1, n_1^k \\ c = \left\{ \begin{array}{ll} \textsf{debranch}(c_1, n, \ell) & \textit{if } n_1 = \textsf{true} \\ \textsf{debranch}(c_2, n, \ell) & \textit{otherwise} \end{array} \right\} \qquad c \mid \mu \Downarrow_{pc} T_2 \mid \mu' \qquad k \sqsubseteq \ell \end{array}}{\textsf{debranch}(\textsf{if } e_1 \textsf{ then } c_1 \textsf{ else } c_2, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \mathtt{DEB-IF-LOW} \rangle \mid \mu'}$$

Rocq, Rocq, Correct!

We choose to prove our theorems and formalise our semantics within the Rocq (formerly Coq) theorem prover.

PartialOrder

```
(*** Formalising Timing Leaks*)

(** * Basic Definitions *)

(** A Partial Order is defined to be a binary relation that is reflexive,
    transitive, but not symmetric. *)
Inductive PartialOrder {A: Type} (rel: A → A → Type) : Type:=
| PartialOrderConstructor
    (rel_refl: ∀ (a: A), rel a a)
    (rel_trans: ∀ (a b c: A), rel a b → rel b c → rel a c)
    (rel_antisym: ∀ (a b: A), a ≠ b → rel a b → rel b a → ⊥).
```

Join Semi-Lattice

```
Inductive Join {A: Type} (rel: A → A → Type) : A → A → A → Type :=
| JoinConstructor
    (pOrderProof: PartialOrder rel)
    (a b join: A)
    (pleft: rel a join)
    (pright : rel b join)
    (pleast: ∀ ub, rel a ub → rel b ub → rel join ub):
  Join rel a b join.

Inductive EX {A: Type} (P: A → Type) : Type :=
| EX_intro (x: A) : P x → EX  P.

(** A Join Semilattice is simply a type equipped with a partial order such that every element has a join. *)
Inductive JoinSemilattice {A: Type} (rel: A → A → Type): Type:=
| JoinSemilatticeConstructor (OrdProof: PartialOrder rel)
    (JoinProof: ∀ (a b: A), EX (λ (join: A) ⇒ Join rel a b join)) .
```

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.
  - reflexivity.
  - reflexivity.
  - specialize (IHe1 _ _ _ _ _ _ _ _ _ _ X1 X0_1).
    specialize (IHe2 _ _ _ _ _ _ _ _ _ _ X2 X0_2).
    rewrite → IHe1.
    rewrite → IHe2.
    reflexivity.
Qed.
```

Observational Equivalence

```
Theorem CommandPreservesMemEq
  {binop_eval: BinOp → Primitive → Primitive → Primitive}
  {rel: Level → Level → Type}
  {latticeProof: JoinSemilattice rel} :
  ∀  {c: Command} {mu1 mu2 mu1' mu2': MemStore} {pc: Level} {T1 T2: TimingList}
     (p1: @CommandBigStep binop_eval rel latticeProof c mu1 pc T1 mu1')
     (p2: @CommandBigStep binop_eval rel latticeProof c mu2 pc T2 mu2')
     (memEq: @MemStoreObservationalEquivalent rel latticeProof mu1 pc mu2),
    @MemStoreObservationalEquivalent rel latticeProof mu1' pc mu2'.
```

Timing Sensitive Non-Interference

```
Theorem CommandTimSec
  {binop_eval: BinOp → Primitive → Primitive → Primitive}
  {rel: Level → Level → Type}
  {latticeProof: JoinSemilattice rel} :
  ∀  {c: Command} {mu₁ mu₂ mu₁' mu₂': MemStore} {pc: Level} {T₁ T₂: TimingList}
     (p₁: @CommandBigStep binop_eval rel latticeProof c mu₁ pc T₁ mu₁')
     (p₂: @CommandBigStep binop_eval rel latticeProof c mu₂ pc T₂ mu₂')
     (memEq: @MemStoreObservationalEquivalent rel latticeProof mu₁ pc mu₂),
    T₁ = T₂.
```

Soundness

```
Theorem CommandSystemSoundness
  {binop_eval: BinOp → Primitive → Primitive → Primitive}
  {rel: Level → Level → Type}
  {latticeProof: JoinSemilattice rel}:
  ∀ {c: Command} {mu mu': MemStore} {T: TimingList} {pc: Level},
    @CommandBigStep binop_eval rel latticeProof c mu pc T mu' →
    @NormalBigStep binop_eval c (StoreProjection mu) (StoreProjection mu').
```

Partial Completeness

```
Theorem CommandSystemCompleteness
  {binop_eval: BinOp → Primitive → Primitive → Primitive}
  {rel: Level → Level → Type}
  {latticeProof: JoinSemilattice rel}:
  ∀ {c: Command} {mu: MemStore} {nu: NormalStore}  {pc: Level}
    (nw: NoWhile c)
    (np: @NormalBigStep binop_eval c (StoreProjection mu) nu),
    EX (λ T ⇒ EX (λ mu' ⇒
       prod (@CommandBigStep binop_eval rel latticeProof c mu pc T mu')
            (StoreProjection mu' = nu))).
```