ISCA'20 Section 5B

Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution

JIYONG YU, NAMRATA MANTRI,

JOSEP TORRELLAS, ADAM MORRISON*, CHRISTOPHER W. FLETCHER

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN *TEL AVIV UNIVERSITY

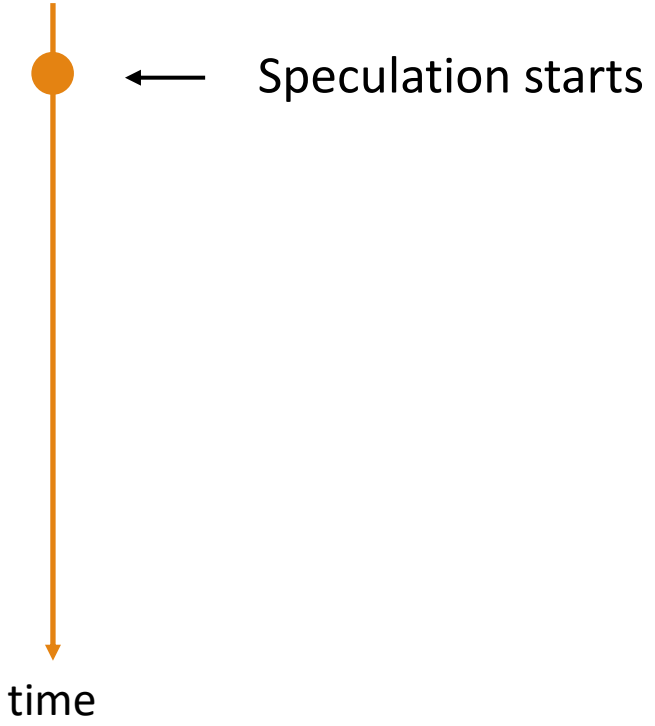
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

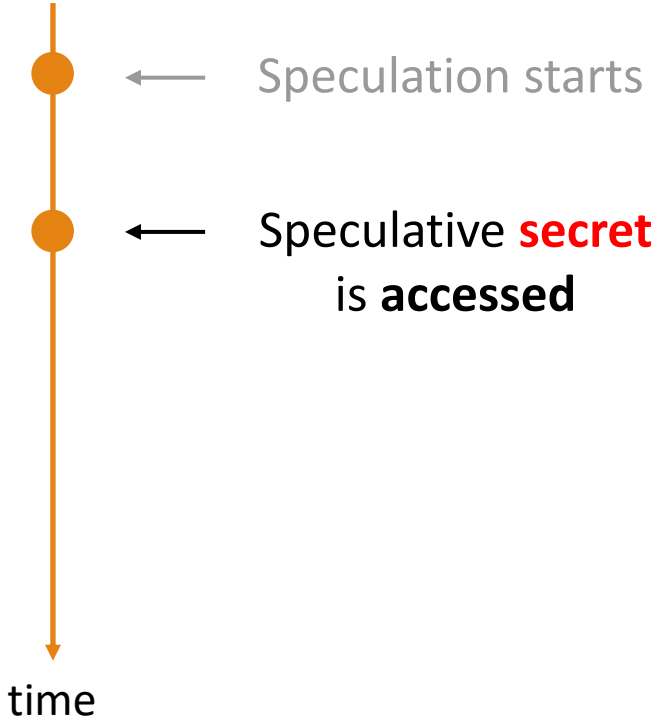
```
if (addr < N) {     // speculation

    // access instruction
    secret = load [addr];

    // transmit instruction
    transmit secret;

}
```



Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

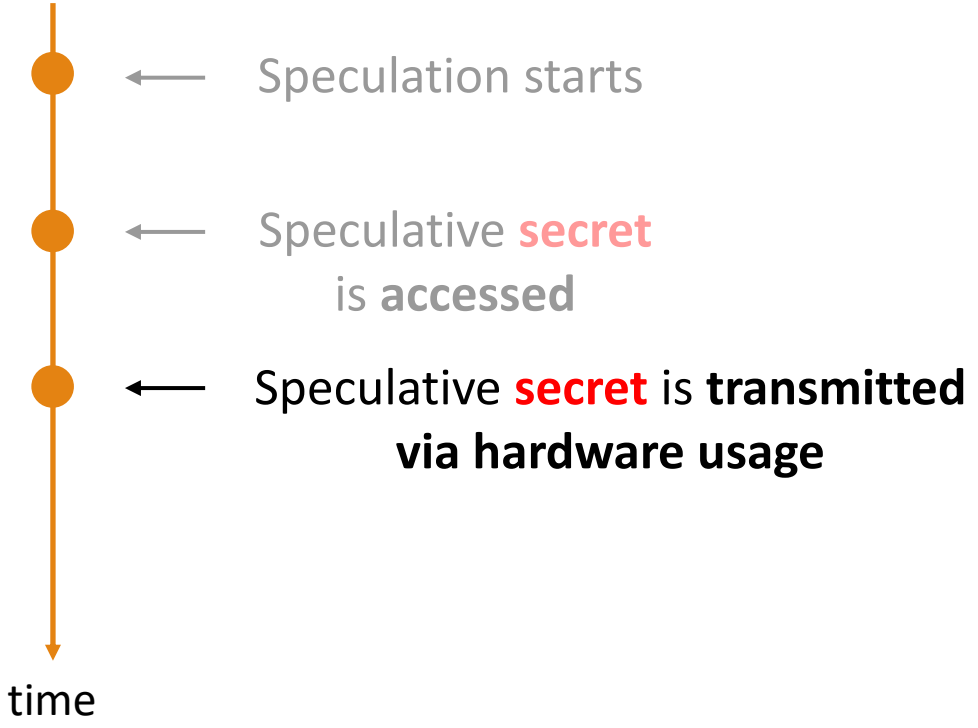
```
if (addr < N) {     // speculation
    // access instruction
    secret = load [addr];

    // transmit instruction
    transmit secret;

}
```



Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```
if (addr < N) {     // speculation

    // access instruction
    secret = load [addr];

    // transmit instruction
    transmit secret;

}
```

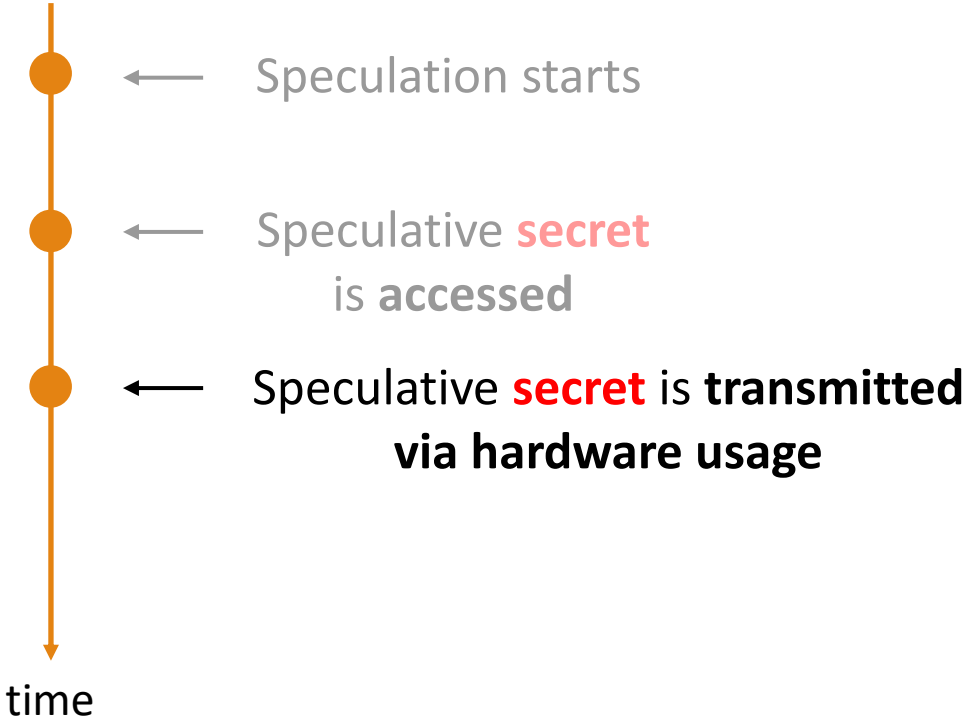


Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```
if (addr < N) {     // speculation

    // access instruction
    secret = load [addr];

    // transmit instruction
    transmit secret;

}
```

Shared hardware



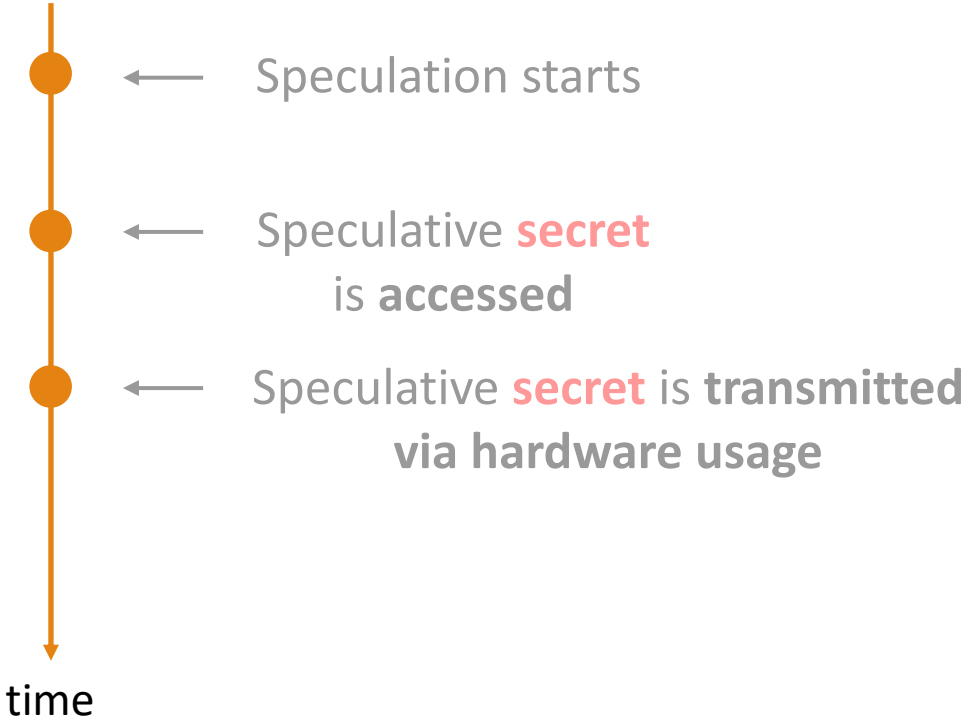
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```
if (addr < N) {    // speculation
    // access instruction
    secret = load [addr];

    // transmit instruction
    transmit secret;
}
```

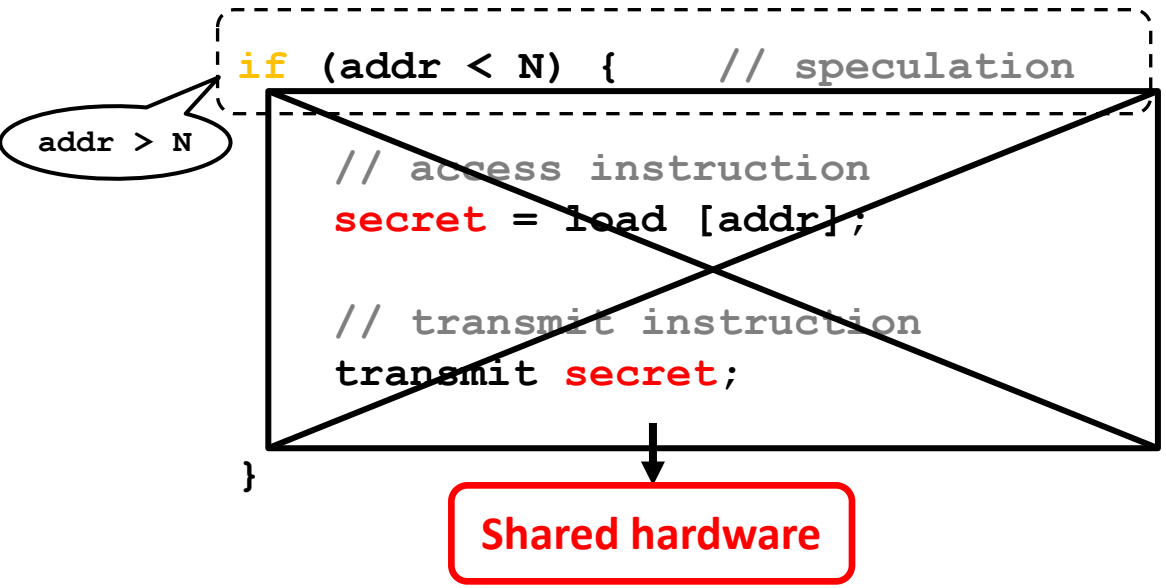
addr < N

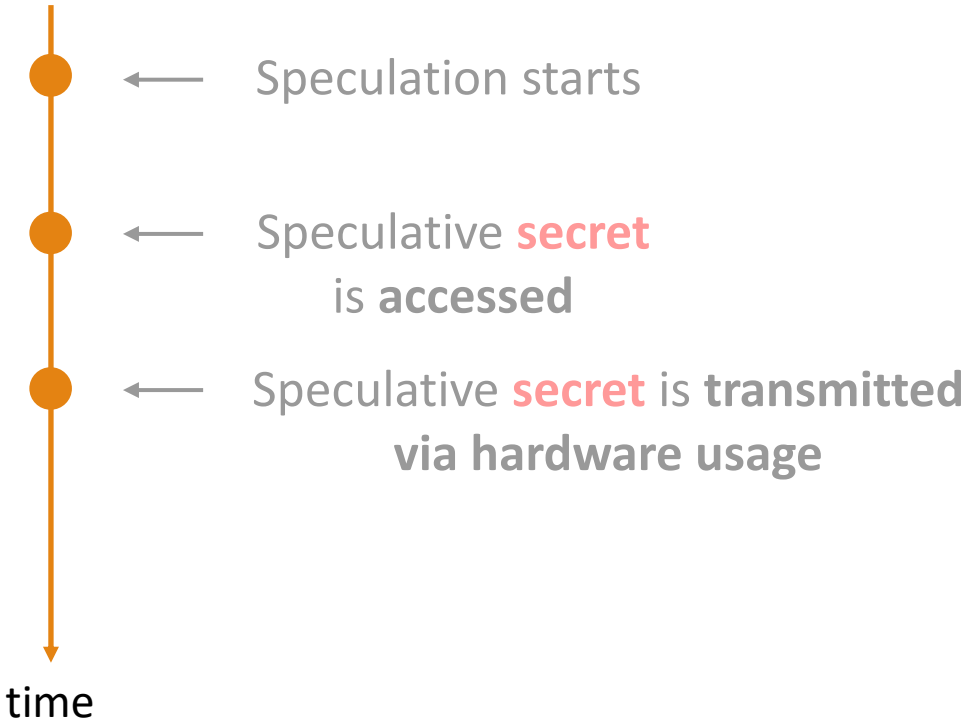
Shared hardware



Speculative Execution Attacks

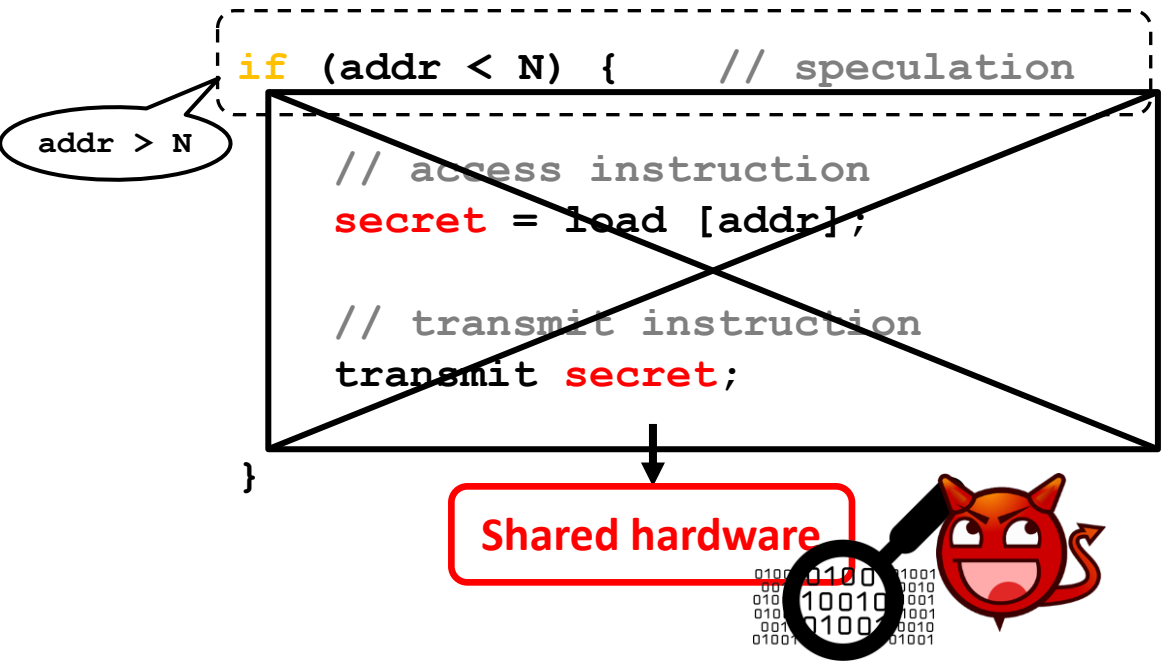
- Attacker exploits **speculative execution** to leak data through hardware usage

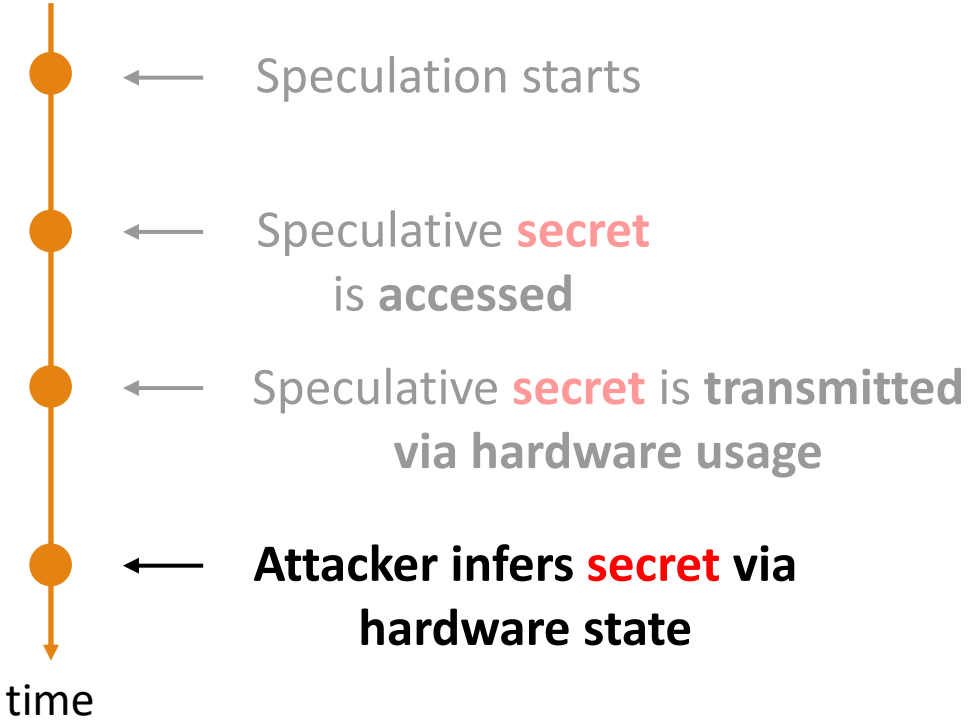




Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage





Existing Mitigations

- How to deal with `transmit secret` ?

Existing Mitigations

- How to deal with `transmit secret` ?
- Solution: Delayed Execution
  - Prior works: SpecShield [PACT'19], NDA [MICRO'19], STT [MICRO'19]

Transmit instruction	Hardware vulnerability
load	Cache side channel
Floating point operations	Subnormal floating point
……	……

Existing Mitigations

- How to deal with `transmit secret` ?
- Solution: Delayed Execution
  - Prior works: SpecShield [PACT'19], NDA [MICRO'19], STT [MICRO'19]

Transmit instruction	Hardware vulnerability
load	Cache side channel
Floating point operations	Subnormal floating point
……	……

```
if (addr < N) {   // speculation

    // access instruction
    secret = load [addr];

    // transmit instruction
    transmit secret;

}
```

→ **Delaying execution**

Existing Mitigations

- How to deal with `transmit secret` ?
- Solution: Delayed Execution
  - Prior works: SpecShield [PACT'19], NDA [MICRO'19], STT [MICRO'19]
- Strong security guarantee
- **High performance overhead**

Transmit instruction	Hardware vulnerability
load	Cache side channel
Floating point operations	Subnormal floating point
……	……

Existing Mitigations

- How to deal with `transmit secret` ?

Register File

Improve the **performance** of *Delayed Execution*

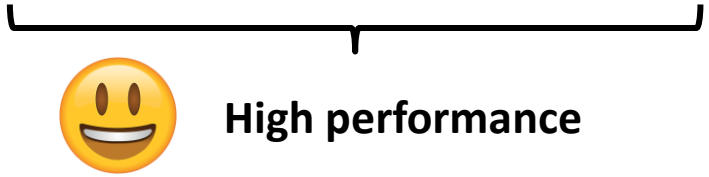
and

Maintain its **security** guarantee

Speculative Data Oblivious (SDO): Executive Summary

Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret`



Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret` by eliminating operand-dependent hardware usage (being data oblivious)

😀 **High performance**

😐 **High security, low performance**

Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret` by eliminating operand-dependent hardware usage (being data oblivious)

Idea 2. Predict how the execution should be performed

Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret` by eliminating operand-dependent hardware usage (being data oblivious)

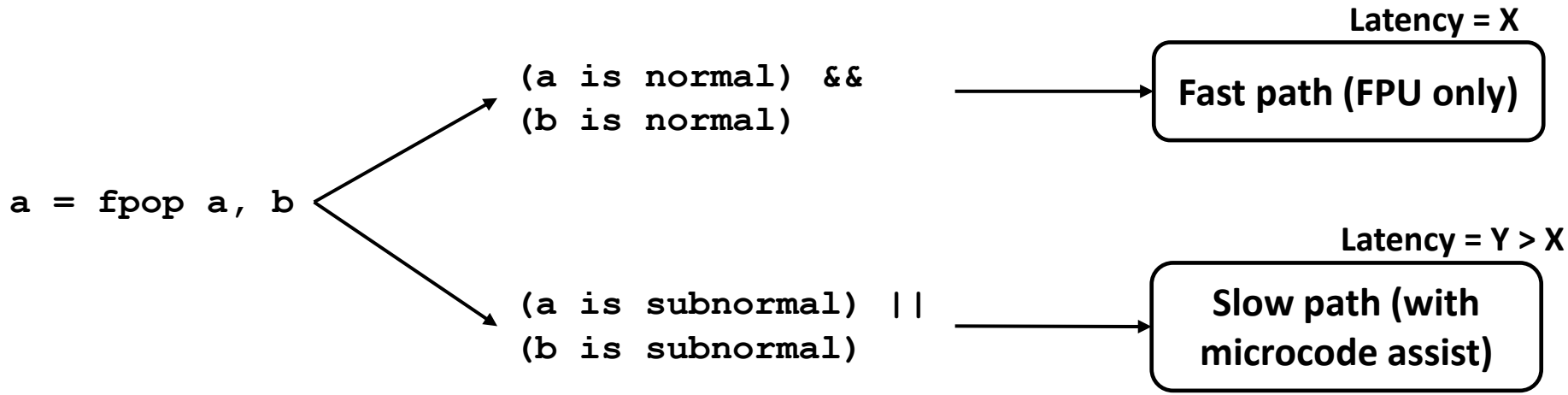
Idea 2. Predict how the execution should be performed

Problem: combining idea 1 & 2 creates security problems

Solution: build on top of Speculative Taint Tracking (STT)

Example: Subnormal Floating-point Operation

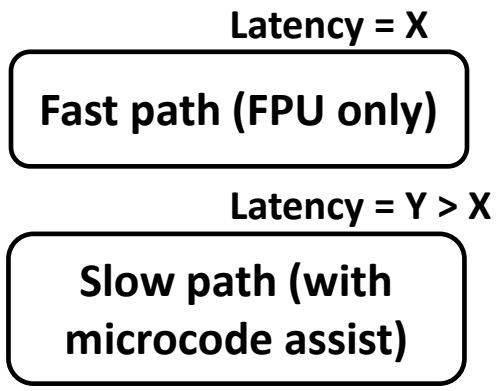
- Double-precision floating point
  - Normal input: (2.23e−308, 1.79e308), processed by Floating-Point Unit (FPU)
  - Subnormal input: (4.9e−324, 2.23e−308), requiring microcode assist



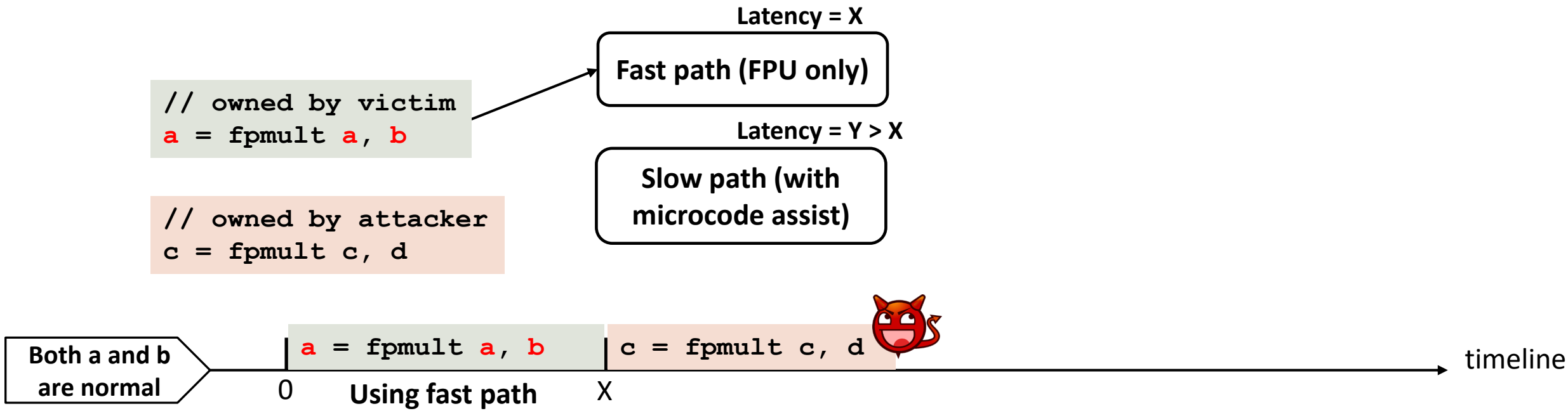
Problem: Leaking Whether Input is Normal/Subnormal

```
// owned by victim
a = fpmult a, b
```

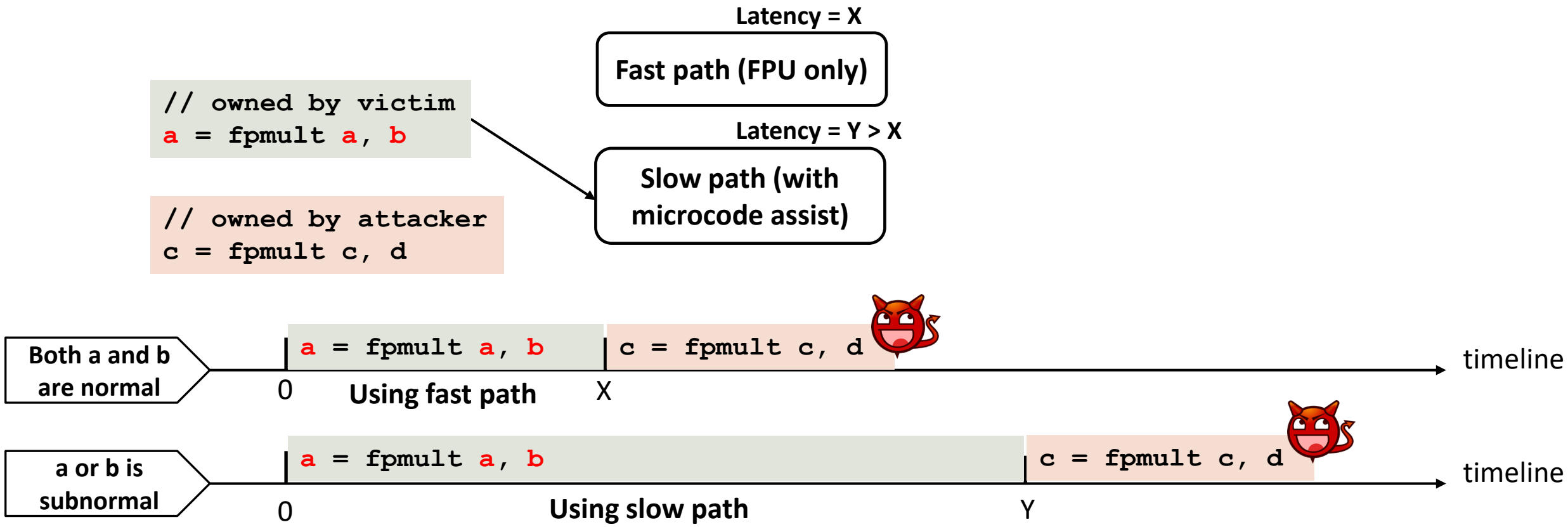
```
// owned by attacker
c = fpmult c, d
```



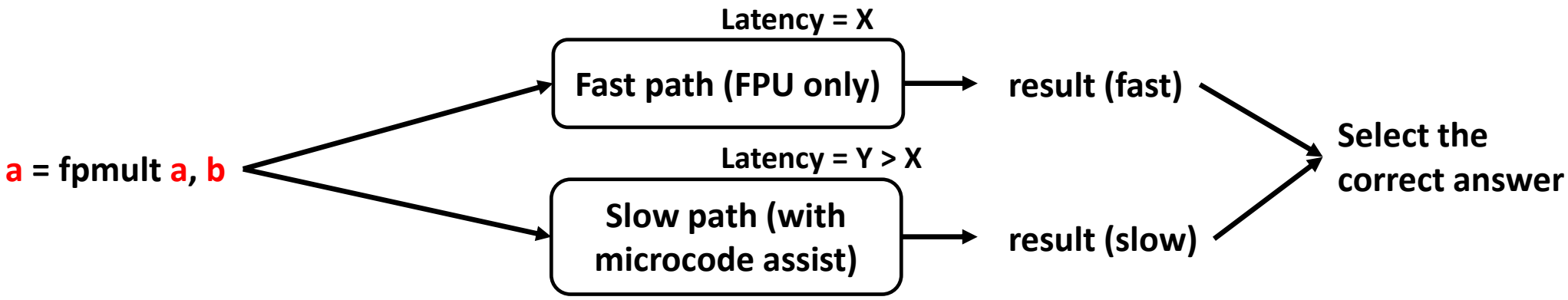
Problem: Leaking Whether Input is Normal/Subnormal



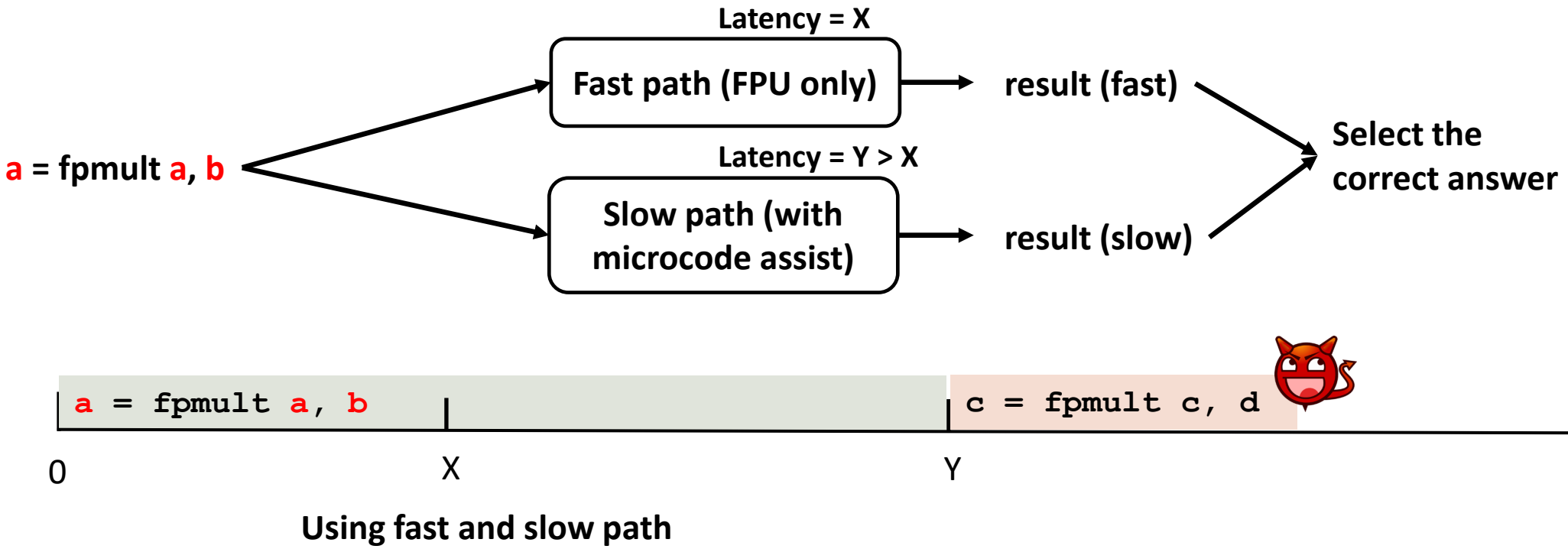
Problem: Leaking Whether Input is Normal/Subnormal



Idea 1: Being Data Oblivious

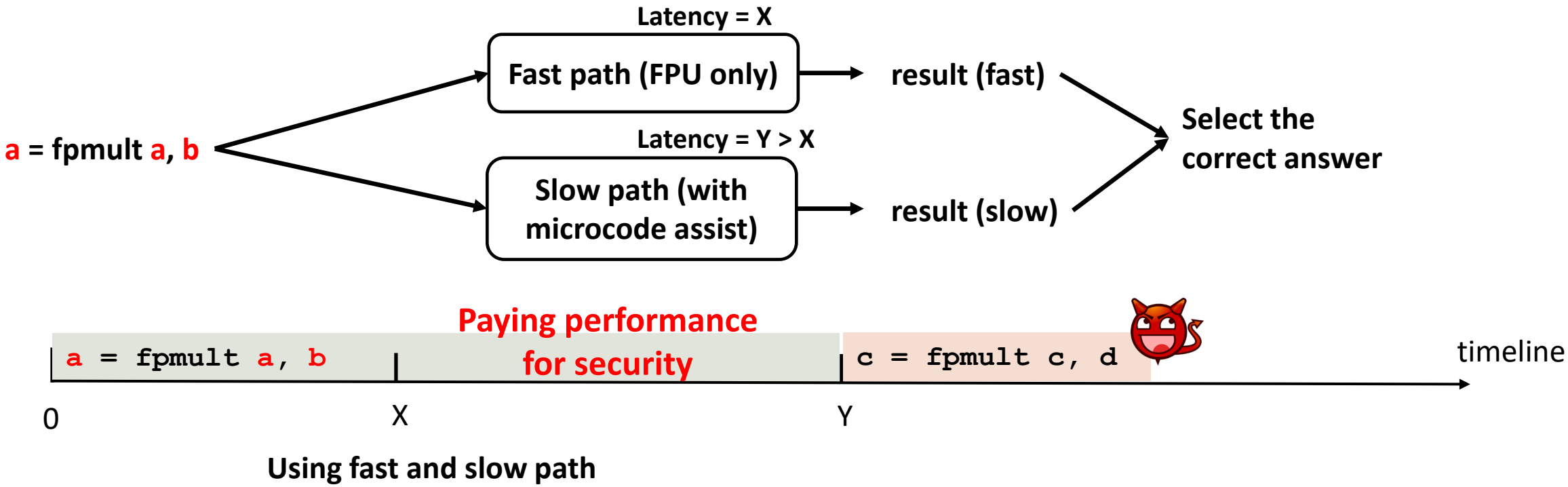


Idea 1: Being Data Oblivious

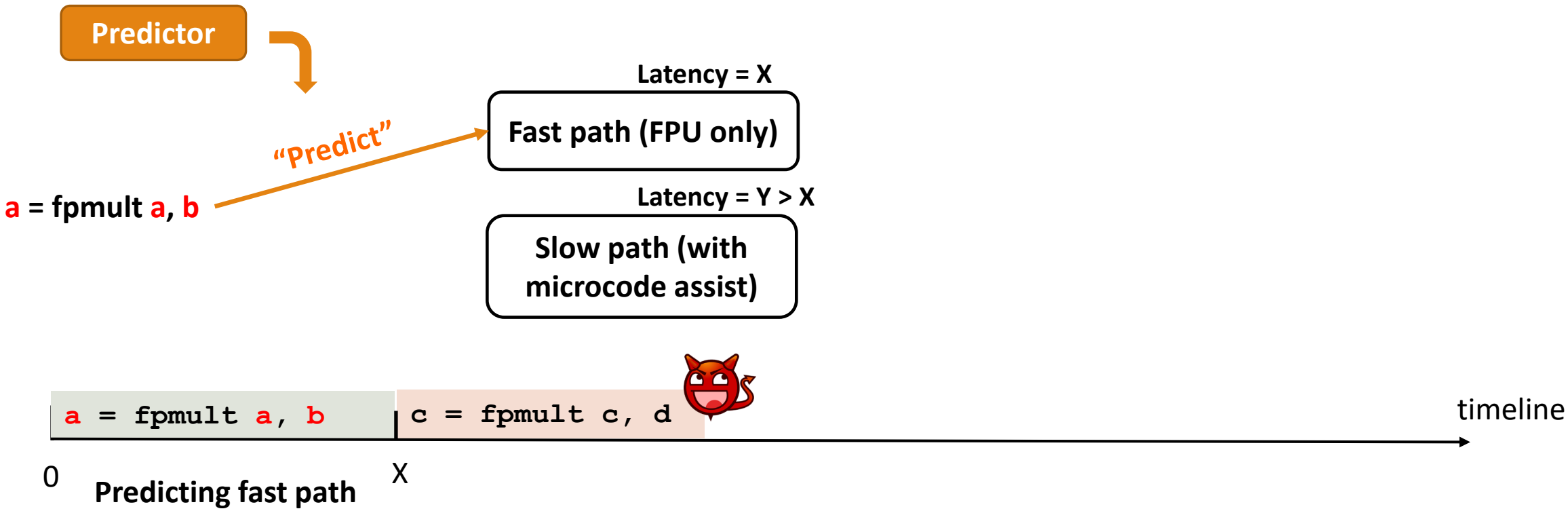


Using fast and slow path

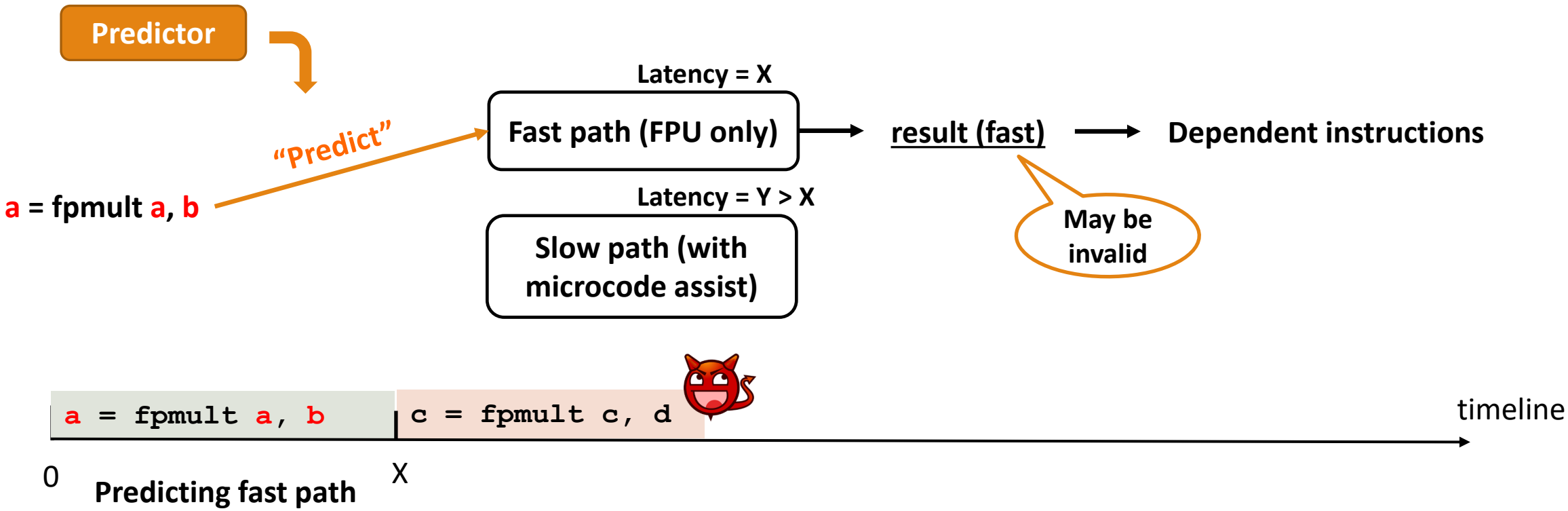
Idea 1: Being Data Oblivious



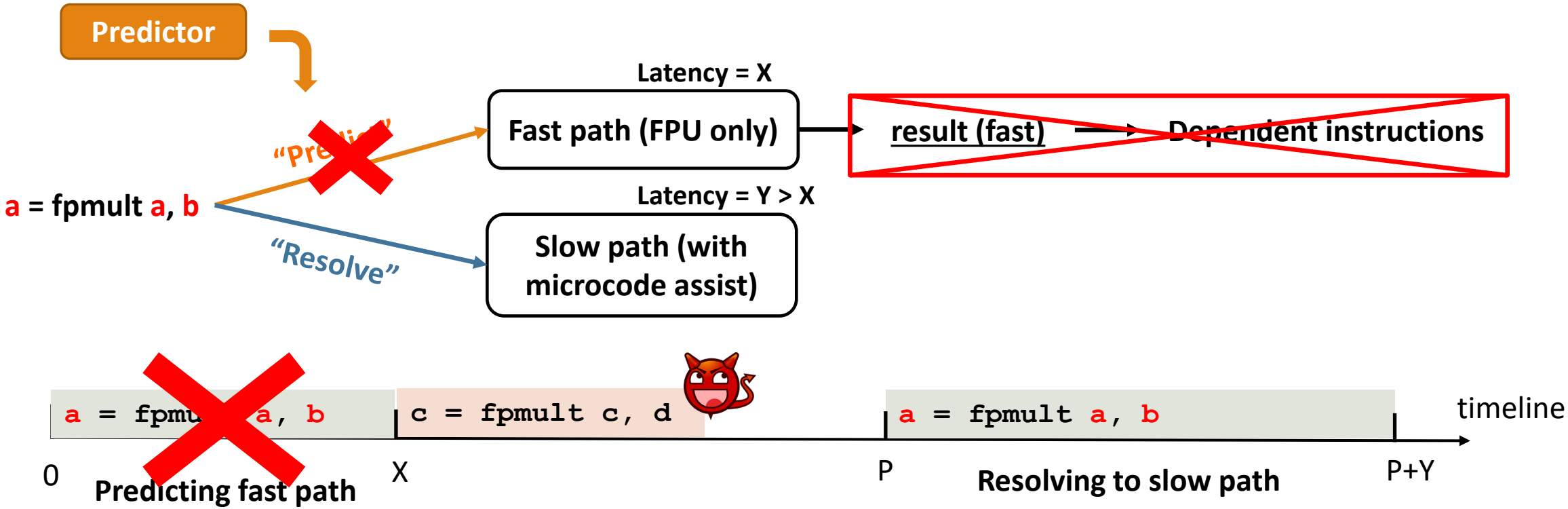
Idea 2: "Predicting" Execution to Perform



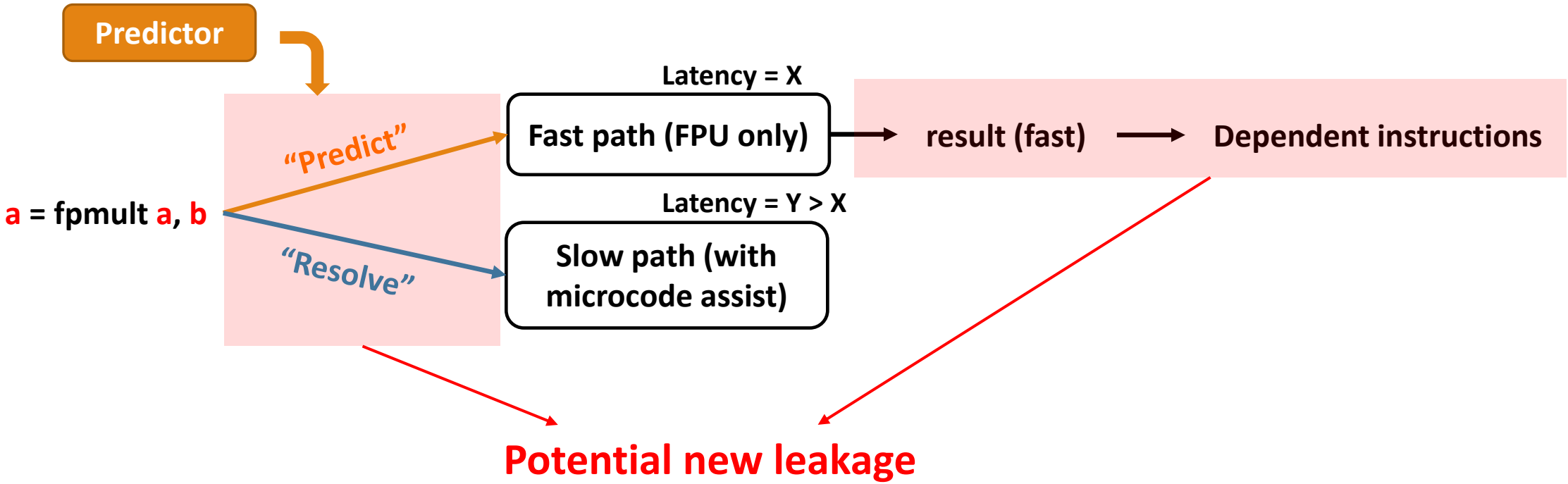
Idea 2: "Predicting" Execution to Perform



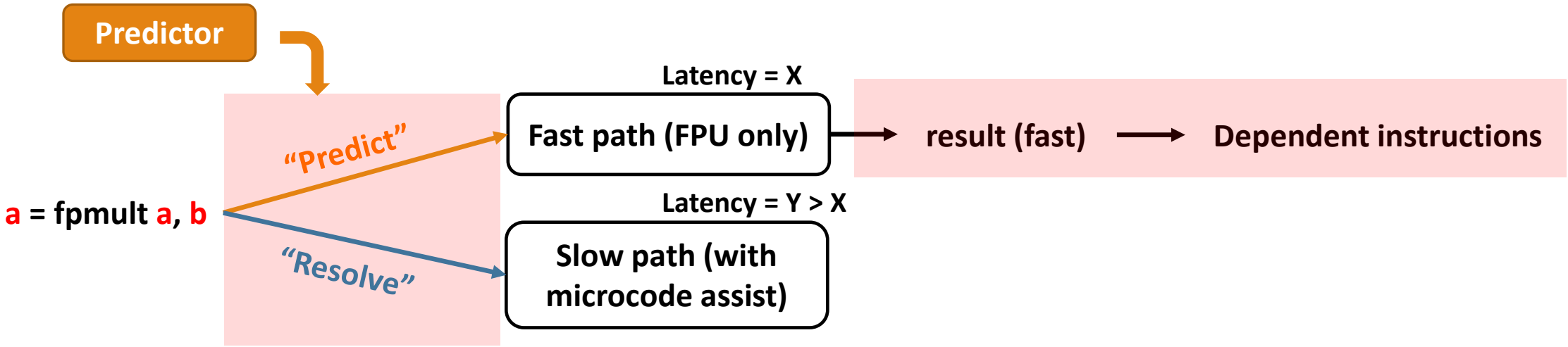
Idea 2: "Predicting" Execution to Perform



Idea 2: "Predicting" Execution to Perform

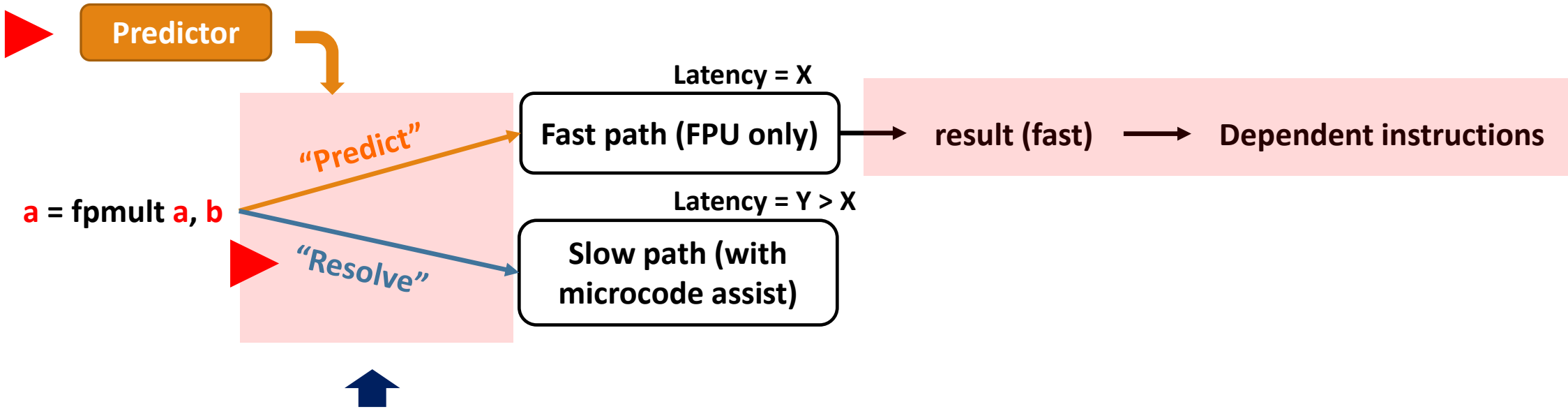


Applying STT for Security

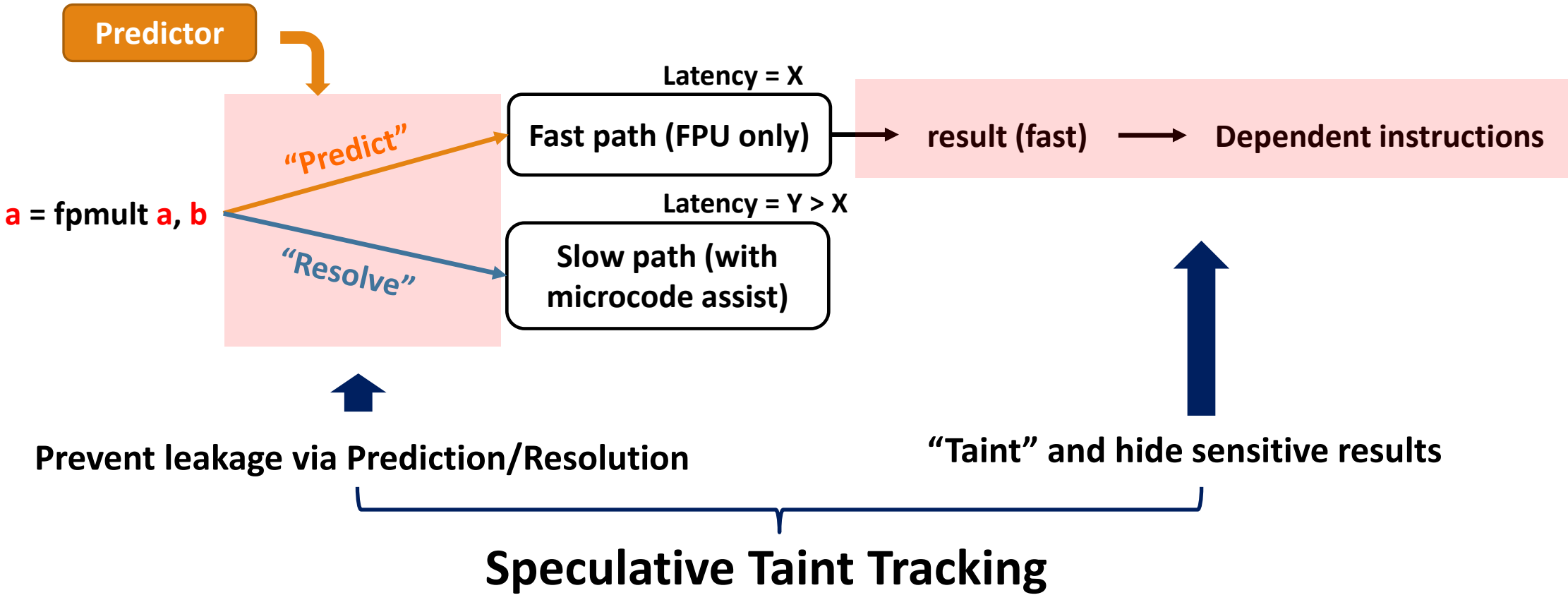


Speculative Taint Tracking

Applying STT for Security



Applying STT for Security



Applying STT for Security

How STT "**prevents leakage via prediction/resolution**":

- Never update predictors with any secret information
- Delay resolution until safe

Applying STT for Security

How STT "**prevents leakage via prediction/resolution**":

- Never update predictors with any secret information
- Delay resolution until safe

How STT "**taints and hides sensitive results**":

- Sensitive data is marked tainted
- Taint propagates through program dataflow
- Transmitters with tainted arguments are handled safely

Applying STT for Security

How STT **prevents leakage via prediction/resolution**

STT Makes Prediction Great (SAFE) Again!

We build predictors to reduce defense overhead

- Taint propagates through program dataflow
- Transmitters with tainted arguments are handled safely

Speculative Data Oblivious Execution (SDO)

Idea 1. Safely execute transmitters in a data-oblivious (DO) manner

Idea 2. Predict how the execution should be performed

Data Oblivious variants + Predicting which variant

+ Safe Prediction with STT

=

SDO

Net result: execute unsafe transmitters early and safely

Speculative Data Oblivious Execution (SDO)

What's Next:

- **Generic SDO Framework**
- **Implementing SDO for load instructions**
- **Evaluation**
- **Conclusion**

SDO Framework

- Step 1: Define data-oblivious (DO) variants for unsafe transmitters

SDO Framework: Step 1: Define Data-oblivious (DO) Variants

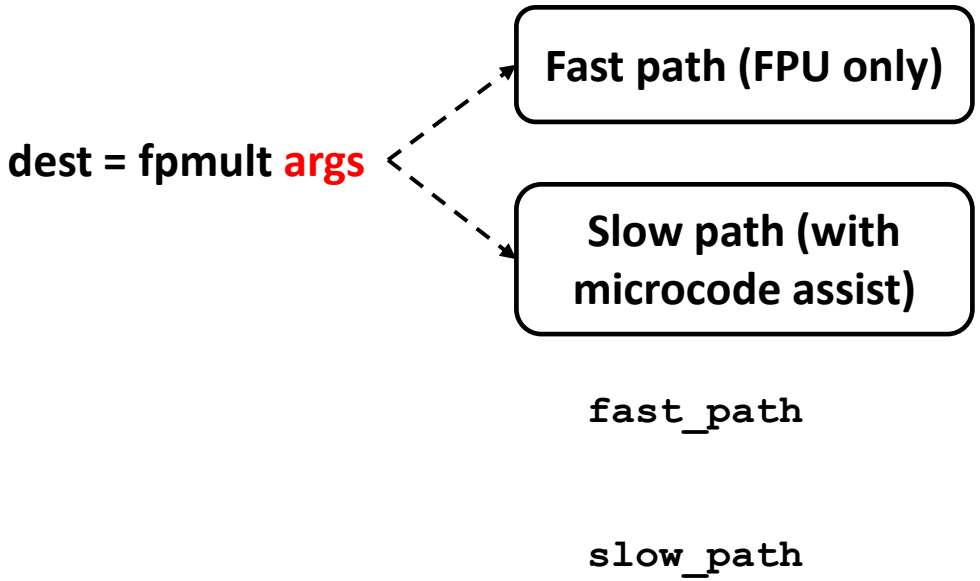
Transmit instruction	`dest <- op args`
DO variants	`DO-op`$_1$ … `DO-op`$_N$

Execution of DO variants	`(dest`$_1$`, success`$_1$`) <- DO-op`$_1$ `args`
	…
	`(dest`$_N$`, success`$_N$`) <- DO-op`$_N$ `args`

SDO Framework: Step 1: Define Data-oblivious (DO) Variants

Transmit instruction	`dest <- op args`
DO variants	`DO-op`$_1$ … `DO-op`$_N$

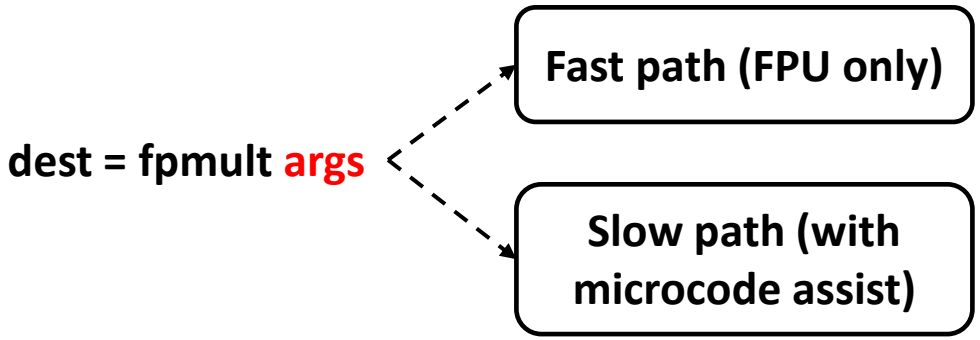
Execution of DO variants	`(dest`$_1$`, success`$_1$`) <- DO-op`$_1$ `args`
	…
	`(dest`$_N$`, success`$_N$`) <- DO-op`$_N$ `args`



SDO Framework: Step 1: Define Data-oblivious (DO) Variants

Transmit instruction	`dest <- op args`
DO variants	`DO-op`$_1$ … `DO-op`$_N$

Execution of DO variants	`(dest`$_1$`, success`$_1$`) <- DO-op`$_1$ `args`
	…
	`(dest`$_N$`, success`$_N$`) <- DO-op`$_N$ `args`



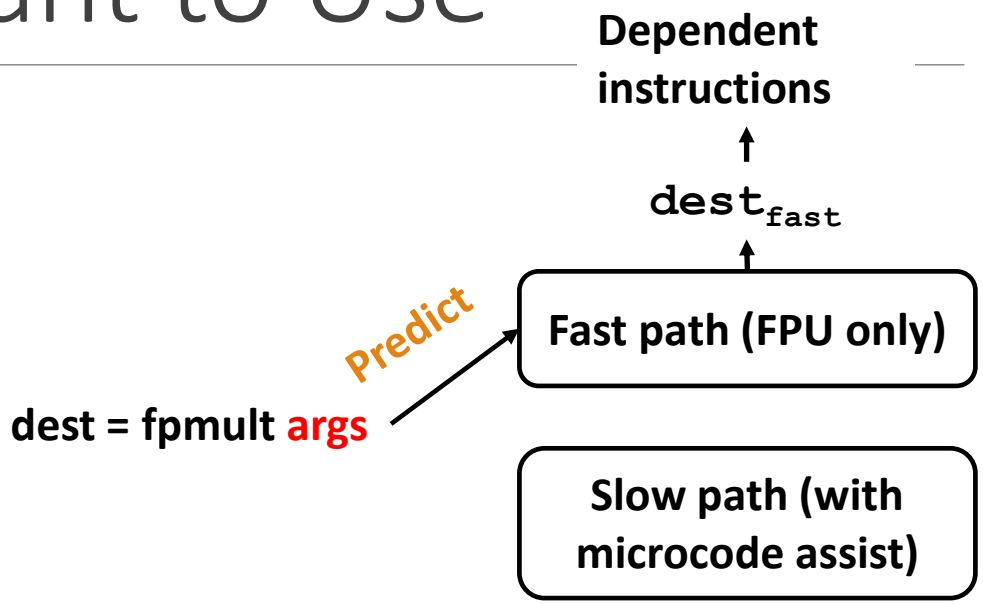

```
dest_fast, success_fast <- fast_path args
  (success_fast = TRUE if args is normal)

dest_slow, success_slow <- slow_path args
  (success_slow = TRUE if args is subnormal)
```

SDO Framework: Step 2: Predict Which DO Variant to Use

Transmit instruction	`dest <- op args`
DO variants	$\texttt{DO-op}_1$ … $\texttt{DO-op}_N$
Predictor	`Pred`
Predicting DO variant	`i <- Pred.predict (public_input)` $(\texttt{dest}_i, \texttt{success}_i) \texttt{ <- DO-op}_i \texttt{ args}$

SDO Framework: Step 2: Predict Which DO Variant to Use

Transmit instruction	`dest <- op args`
DO variants	`DO-op`$_1$ … `DO-op`$_N$
Predictor	`Pred`
Predicting DO variant	`i <- Pred.predict (public_input)` `(dest`$_i$`, success`$_i$`) <- DO-op`$_i$ `args`



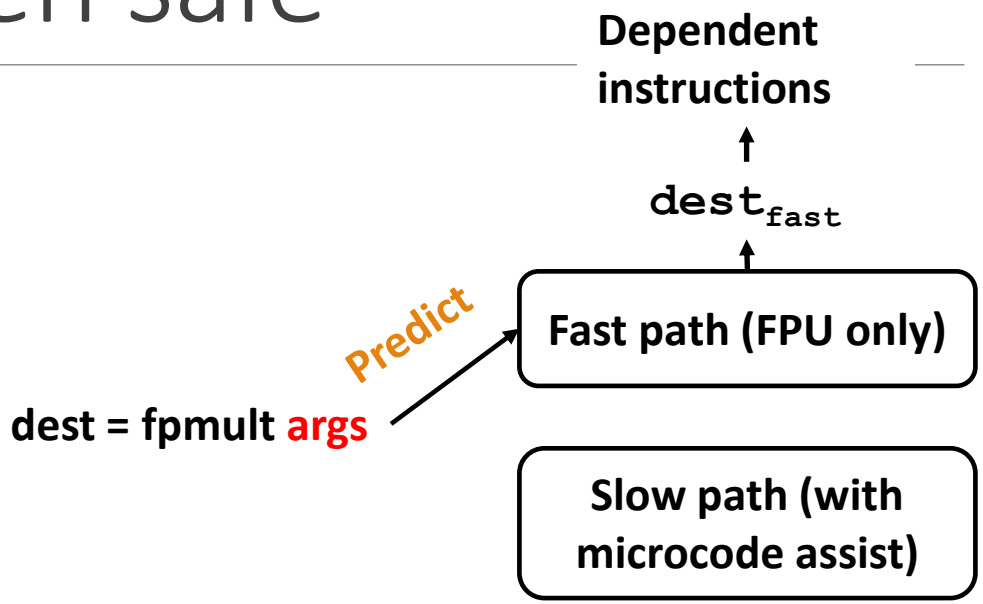
Static Predictor: always predicting "Fast path"

```
dest_fast, success_fast <- fast_path args
  (success_fast = TRUE if args is normal
   success_fast = FALSE if args is subnormal)
```

SDO Framework: Step 3: Resolve Prediction when safe

Transmit instruction	`dest <- op args`
DO variants	`DO-op`$_1$ … `DO-op`$_N$
Predictor	`Pred`
Predicting DO variant	`i <- Pred.predict (public_input)` `(dest`$_i$`, success`$_i$`) <- DO-op`$_i$ `args`

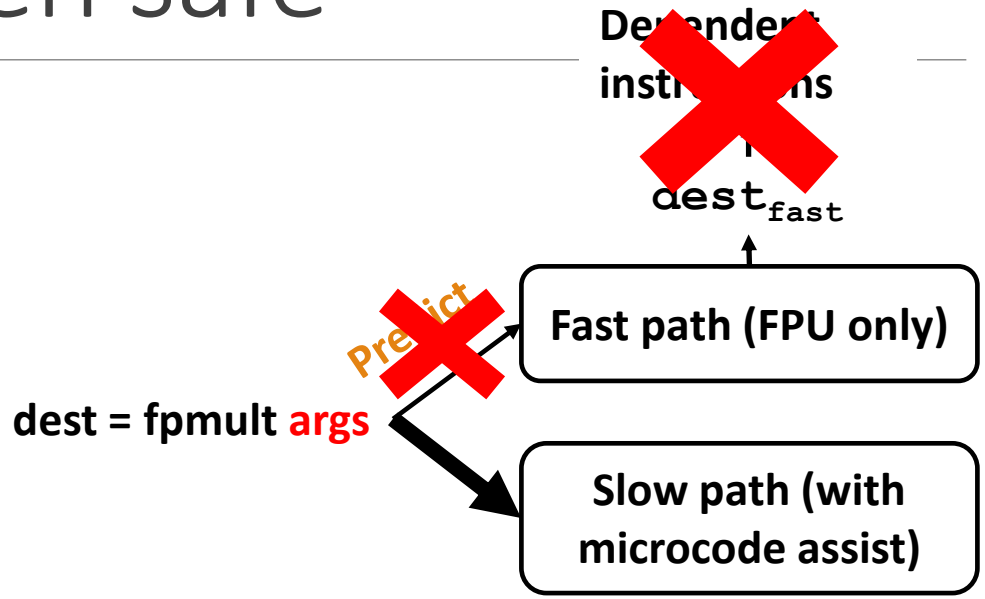
Resolving when safe	`Pred.update(…)`
	`if (!success`$_i$`)`
	`  squash from "dest <- op args"`



SDO Framework: Step 3: Resolve Prediction when safe

Transmit instruction	`dest <- op args`
DO variants	`DO-op`$_1$ … `DO-op`$_N$
Predictor	`Pred`
Predicting DO variant	`i <- Pred.predict (public_input)` `(dest`$_i$`, success`$_i$`) <- DO-op`$_i$ `args`

Resolving when safe	`Pred.update(…)`
	`if (!success`$_i$`)`
	`squash from "dest <- op args"`



success$_{fast}$ == FALSE

Designing SDO for Loads

- Load is the vital motivation and challenge for SDO
  - The execution of loads is complicated, susceptible to various attacks
  - Most performance overhead comes from loads

Step 1: Define DO Variants for Loads

- DO variants
  - **DO-ld$_{L1}$** : only accessing L1
  - **DO-ld$_{L2}$** : only accessing L1 and L2 sequentially
  - **DO-ld$_{L3}$** : only accessing L1, L2 and L3 sequentially
  - **DO-ld$_{Mem}$** : accessing L1, L2, L3 and DRAM sequentially
- **(dest$_{xx}$, success$_{xx}$) <- DO-ld$_{xx}$ addr** **// dest$_{xx}$ = ⊥ if success$_{xx}$ == FALSE**

Step 1: Define DO Variants for Loads

- DO variants
  - **DO-ld$_{L1}$** : only accessing L1
  - **DO-ld$_{L2}$** : only accessing L1 and L2 sequentially
  - **DO-ld$_{L3}$** : only accessing L1, L2 and L3 sequentially
  - **DO-ld$_{Mem}$** : accessing L1, L2, L3 and DRAM sequentially
- **(dest$_{xx}$, success$_{xx}$) <- DO-ld$_{xx}$ addr // dest$_{xx}$ = ⊥ if success$_{xx}$ == FALSE**
- DO variants (**DO-ld$_{Li}$**) must be free of adversary-observable hardware resource usage
  - Cannot modify cache state (tag, data, LRU bits, etc.)
  - Cannot incur address-dependent latency (e.g., free of bank conflict, port contention)
  - ……

Step 1: Define DO Variants for Loads

- DO variants
  - **DO-ld$_{L1}$** : only accessing L1
  - **DO-ld$_{L2}$** : only accessing L1 and L2 sequentially
  - **DO-ld$_{L3}$** : only accessing L1, L2 and L3 sequentially
  - **DO-ld$_{Mem}$** : accessing L1, L2, L3 and DRAM sequentially
- **(dest$_{xx}$, success$_{xx}$) <- DO-ld$_{xx}$ addr** **// dest$_{xx}$ = ⊥ if success$_{xx}$ == FALSE**
- DO variants (**DO-ld$_{Li}$**) must be free of adversary-observable hardware resource usage
  - Cannot modify cache state (tag, data, LRU bits, etc.)
  - Cannot incur address-dependent latency (e.g., free of bank conflict, port contention)
  - ……
- For more details (e.g., load re-ordering, performance optimizations) about DO variants, please see the paper

Step 2: Predict Which DO Variant to Use

- Goal: **accurate** and **precise** cache level prediction
  - Suppose a load requires data from cache level `i` and the predictor predicts level `j`
  - "accurate and precise": `i == j`
  - "accurate but imprecise": `i < j` -> redundant cache access -> unnecessary load latency
  - "inaccurate": `i > j` -> cache miss -> writeback ⊥ to dependents -> squash

Predicted level	DO Variant
1 (L1)	DO-ld$_{L1}$
2 (L2)	DO-ld$_{L2}$
3 (L3)	DO-ld$_{L3}$
4 (Memory)	DO-ld$_{Mem}$

Step 2: Predict Which DO Variant to Use

- Goal: **accurate** and **precise** cache level prediction
  - Suppose a load requires data from cache level `i` and the predictor predicts level `j`
  - "accurate and precise": `i == j`
  - "accurate but imprecise": `i < j` -> redundant cache access -> unnecessary load latency
  - "inaccurate": `i > j` -> cache miss -> writeback ⊥ to dependents -> squash
- Hybrid predictor:
  - "Greedy" (for loads with irregular access pattern):

    Maintain a history, and pick the lowest level among history
  - "Loop" (for loads with regular access pattern)

    Learn the recurring pattern, and predict based on the pattern

Predicted level	DO Variant
1 (L1)	DO-ld$_{L1}$
2 (L2)	DO-ld$_{L2}$
3 (L3)	DO-ld$_{L3}$
4 (Memory)	DO-ld$_{Mem}$

Step 3: Resolve When Load is Safe

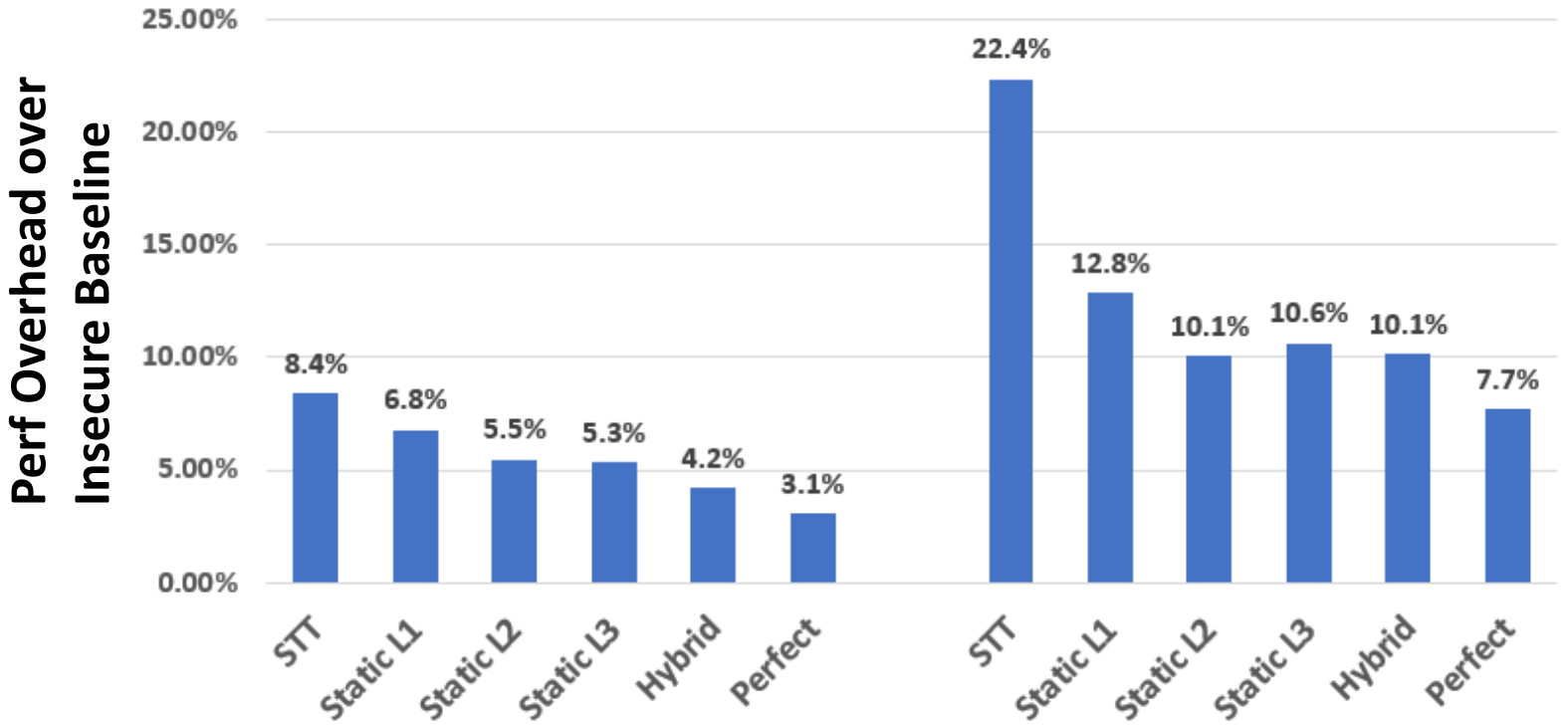
- Update the predictor
- Squash if **`success == FALSE`**
- In a multi-processor:

  `DO-ld`$_{Lx}$ cannot modify cache state

  → Data fetched by **`DO-ld`**$_{Lx}$ may not be cached in L1

  → May missing cache invalidation
- Solution: send a second load request to validate if a cache invalidation was missed
  - We adopt the validation infrastructure proposed in InvisiSpec [MICRO'18]

Performance Evaluation on SPEC2017



"Spectre" attack model
Consider control-flow speculation

"Futuristic" attack model
Consider all types of speculation

Transmitters:
- **Load**
- **Floating-point multiplication**
- **Floating-point division**

Static L1: always predicting DO-ld_{L1}
Static L2: always predicting DO-ld_{L2}
Static L3: always predicting DO-ld_{L3}
Hybrid: using the hybrid predictor
Perfect: prediction is accurate and precise

Conclusion

- SDO serves as a new speculative execution attack mitigation with high-performance and high-security
- The proposed SDO framework augments STT with significant speedup without compromising security

Data Oblivious variants + Predicting which variant + Safe Prediction with STT

=

Safe, early execution of transmitters

Applying STT for Security

STT: prediction and resolution never depend on sensitive data

We can build new predictors to get more performance

Prevent leakage via Prediction/Resolution

“Taint” and hide sensitive results

Speculative Taint Tracking