Systematic Timing Leakage Analysis of NIST PQDSS Candidates: Tooling and Lessons Learned

Olivier Adjonyo
Emanuele Bellini
Gilbert Ndollane Dione
Robert Merget

Technology Innovation Institute
Abu Dhabi, UAE

Sébastien Bardin
Mahmudul Faisal Al Ameen
Frédéric Recoules
Yanis Sellami

Université Paris-Saclay, CEA List
Palaiseau, France

The Journey

The Leak - timing side-channels and why constant-time programming

The Blind Spot - binaries and why manual checks fail

The Weapon - automatic, multidimensional evaluation

The Campaign - systematic experiments across primitives

The Discoveries - interesting case studies

The Verdict - recommendations for real-world PQC

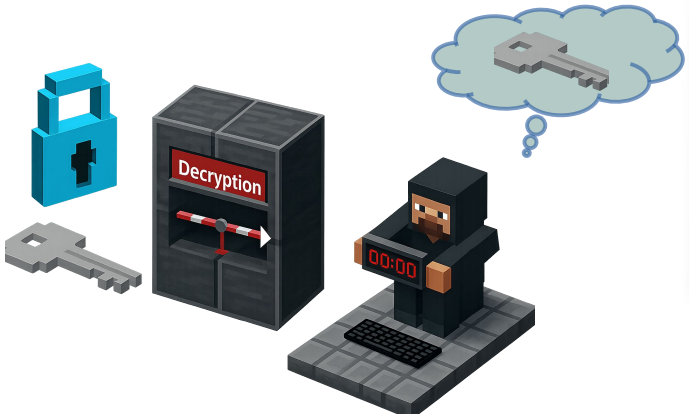
22 violations reported

5 fixes in 4 implementations

58 alerts still to process

Timing Side-channel for Cryptography

Timing Side-channel Attack



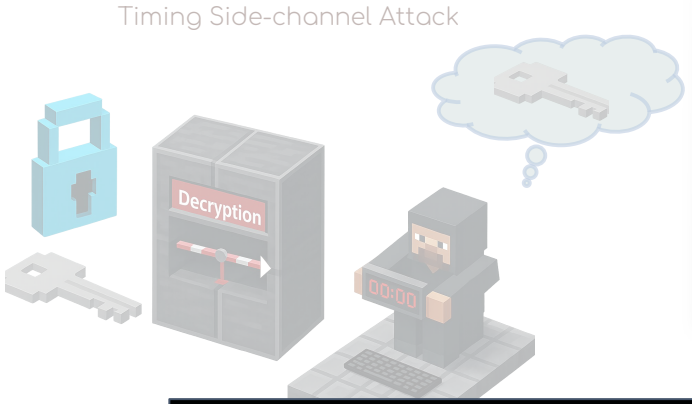
Execution time differences
leaks exponent bits

- **Kocher (1996) →**
  - Introduced the square-and-multiply attack on RSA.
- **Boneh et al. (2006) →**
  - Emphasized need for blinding and constant-time implementations.
- **Primas et al. (2017) →**
  - Side-Channel Attacks on Masked Lattice-Based Encryption
- **Park et al. (2018) →**
  - Side-Channel Attacks on Multivariate Quadratic Equations

Quantum security does not ensure safety from side-channels

So, resistance from timing-attack is important in PQC

Timing Side-channel for Cryptography



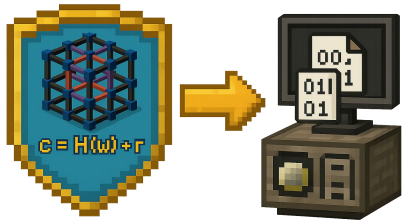
- Kocher (1996) →
  - Introduced the square-and-multiply attack on RSA.
- Boneh et al. (2006) →
  - Emphasized need for blinding and constant-time implementations.
- Primas et al. (2017) →
  - Side-Channel Attacks on Masked Lattice-Based Encryption
- Park et al. (2018) →
  - Side-Channel Attacks on Multivariate Quadratic Equations

Constant-time programming ensures a program is resistant to timing side-channels

The Stakes: Why Binary-Level Analysis Matters

- Unintentional Mistakes of Programmers
- Use of Non-constant-time Library Functions
- Compiler Introduced Leak

Real-World Deployment: **Binary**



CT analysis isn't enough in source level

```
uint8_t x =
((a == 0) && b)||
((a == 1) && c);
```

Compilation

Breaks CT Property

```
 4: test %dil,%dil
 7: jne 18
 9: mov $0x1,%eax
 e: test %sil,%sil
11: je 18
```

The Stakes: Why Automatic Analysis Matters

Manual Analysis:

Scalability Challenge



Necessary Approach:

Automatic Evaluation Process



The Gap: Existing Tool Limitations

Static

CacheAudit FlowTracker
BLAZER CT-LLVM

Dynamic

ctgrind MicroWalk
DATA TIMECOP

Symbolic

CacheD Binsec/Rel

Statistical

dudect RTLF
mona tlsfuzzer

CT Violation

+ Prove absence of vulnerabilities
- Cannot generate witness
- High false positive rate

CT Violation

+ Explores deep trace
- False positive (taint analysis)
- May miss the leaking path

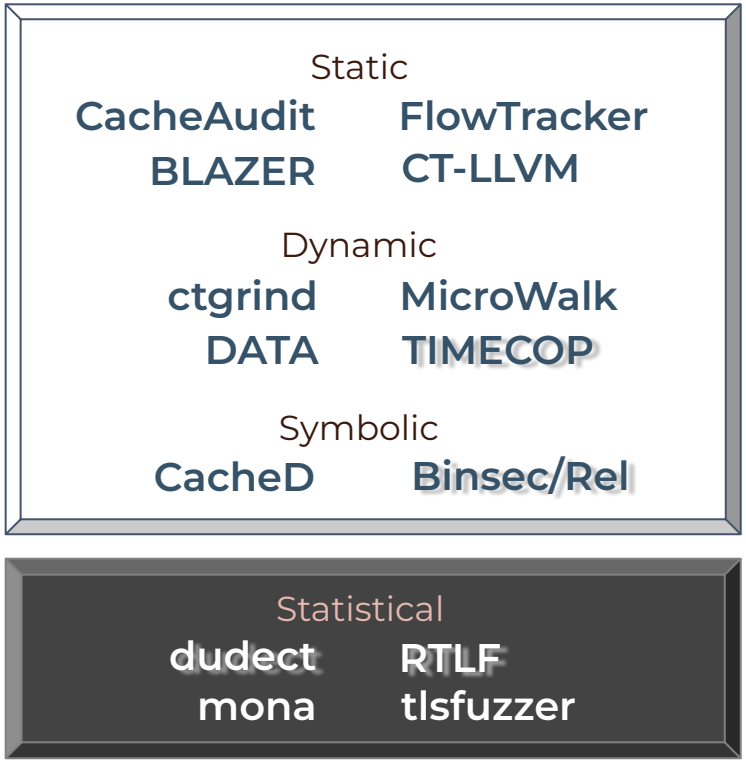
CT Violation

+ Higher precision
- Scalability is often limited in practice

Timing Leak

+ Can report real timing leaks
- Cannot perform fault localization

The Gap: Limitations in Usability of the Tools



- **Complex setup procedures** requiring deep tool-specific knowledge
- Inconsistent **input formats** and **configuration** requirements
- Difficult **result interpretation** with high false positive rates
- Limited **cross-tool comparison** capabilities

Chosen Tools with Complementary Strength

Static

CacheAudit FlowTracker

BLAZER CT-LLVM

Dynamic

ctgrind MicroWalk

DATA **TIMECOP**

Symbolic

CacheD **Binsec/Rel**

Statistical

dudect **RTLF**

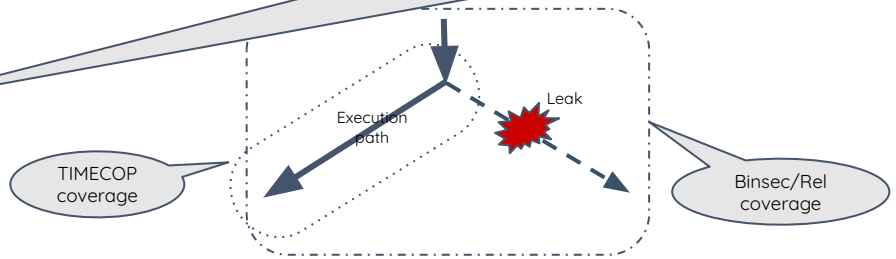
mona tlsfuzzer

Deeper Analysis vs. High False Positive

Control flow leak and Memory access leak

High Precision vs. Bounded Analysis

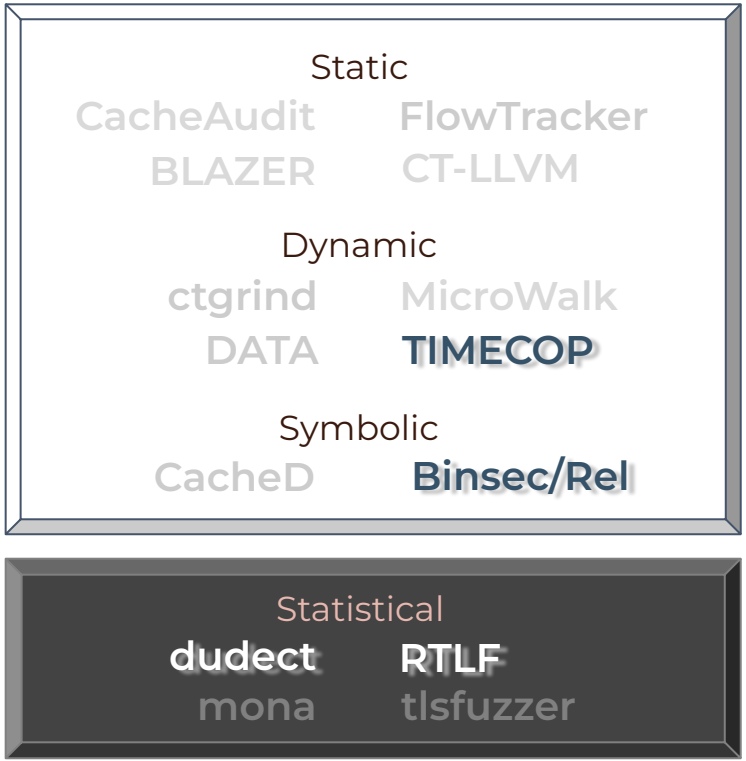
Control flow leak, Memory access leak, Multiplication leak, and Division leak



True evidence of leakage vs. No Localization

Existence of (any) timing side-channel leak

Filling the Gap: Automated Evaluation Process



We needed a **turnkey, consistent pipeline** that could systematically evaluate timing leakage across all NIST PQC candidates.

Challenges of An Automated Evaluation Process

Test Harness
Standardized test harnesses target crypto_sign() entry points with consistent secret input tainting

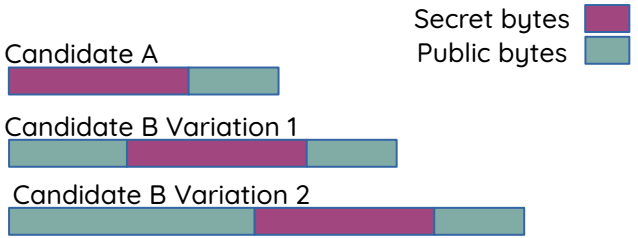
Builds and Compiles
Automated compilation for various variations of candidates

Analysis (4 Tools)
Automatic analysis by TIMECOP, Binsec/Rel, dudect, and RTLF analysis engines

Aggregate Results
Unified reporting with vulnerability classification and actionable findings

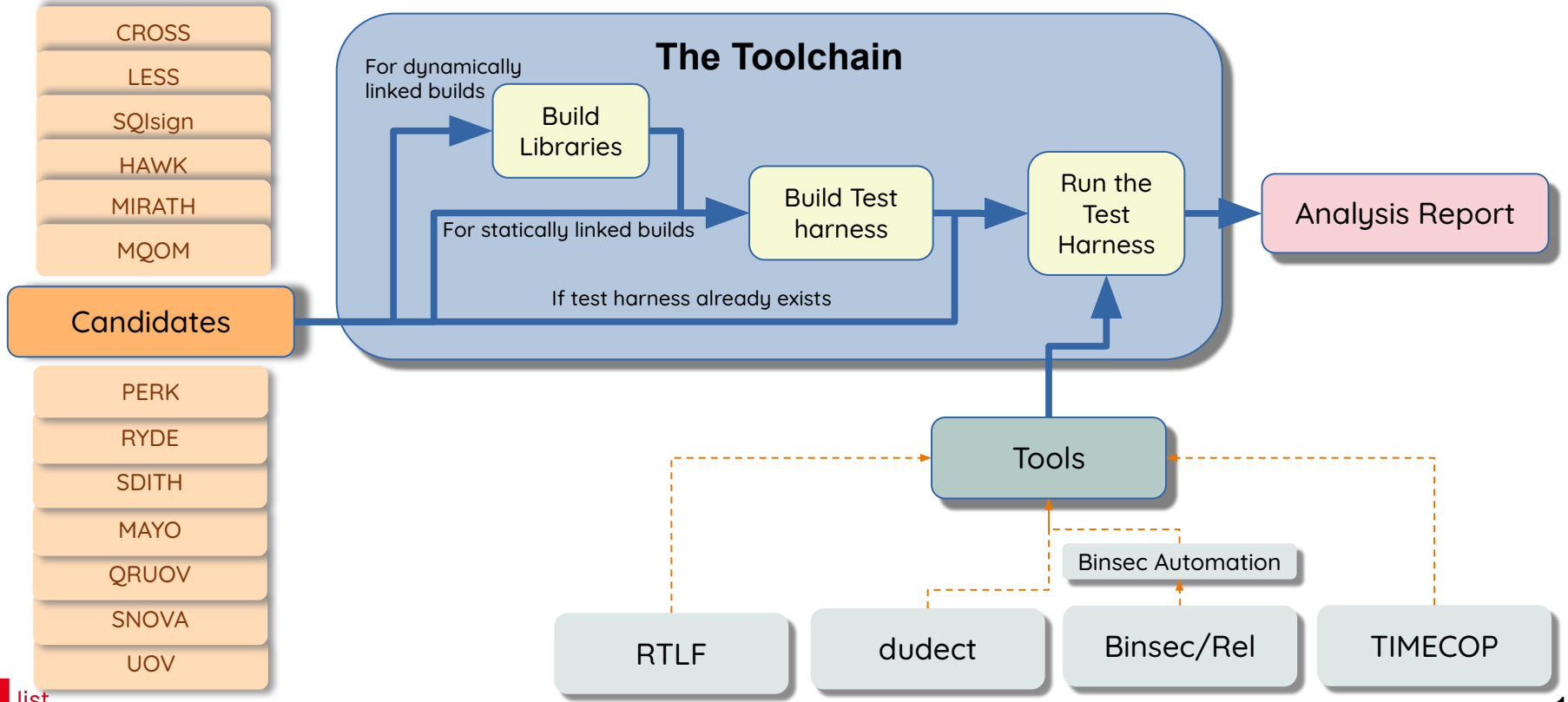
Example of a challenge

Secret bytes component placement in a secret key is inconsistent among various candidates



Needs standardization for automated checking

Automated Evaluation Process: Workflow



Experiments & Outcomes

15

Primitives Covered

Major post-quantum signature schemes from NIST standardization rounds 1 and 2

202

Instances Analyzed

Comprehensive evaluation across all of the available configurations of round 1 and round 2 implementations

22

Alerts Reported

Confirmed CT violations identified and documented (58 still in investigation)

5

Fixes Achieved

Successful remediation through developer collaboration (2 still in progress)

TIMECOP

134

Alerts

134 unique source-level alerts are identified from 7909 binary-level alerts

57.9%

False Positive

Manual estimation on a subset of 76 alerts

Binsec/Rel

22

Alerts

22 unique source-level alerts are identified from 465 binary-level alerts

0%

False Positive

22 alerts are manually checked and no false positive is found

Reported and Fixed

SNOVA

PREON

LESS

MIRATH

Results: At A Glance

		Binsec/Rel	dudect	RTLF	TIMECOP
R1	**SNOVA**	Critical	-	-	Alerts Not Checked
	PREON	Critical	-	-	Critical
R2	**CROSS**	Non-critical	Secure*	Secure*	Non-critical, False Positive
	LESS	Non-critical	Secure*	Insecure	Critical, False Positive
	MIRATH	Critical	Secure*	Insecure	Critical
	HAWK	CT Violation	Secure*	Insecure	Non-critical
	PERK	Non-critical	Secure*	Insecure	Non-critical, False Positive
	RYDE	CT Violation	Insecure	Insecure	Non-critical
	SNOVA	Non-critical	-	-	Alerts Not Checked
	MAYO	Non-critical	Secure*	Secure*	False Positive
	MQOM	Secure*	Secure*	Secure*	False Positive
	QRUOV	Non-critical	Secure*	Secure*	Non-critical
	SDITH	Secure*	Secure*	Secure*	Alerts Not Checked
	SQIsign	Secure*	Secure*	Insecure	Alerts Not Checked
	UOV	Secure*	-	-	Alerts Not Checked

Legends for primitives:

Exploitability is confirmed and fixed

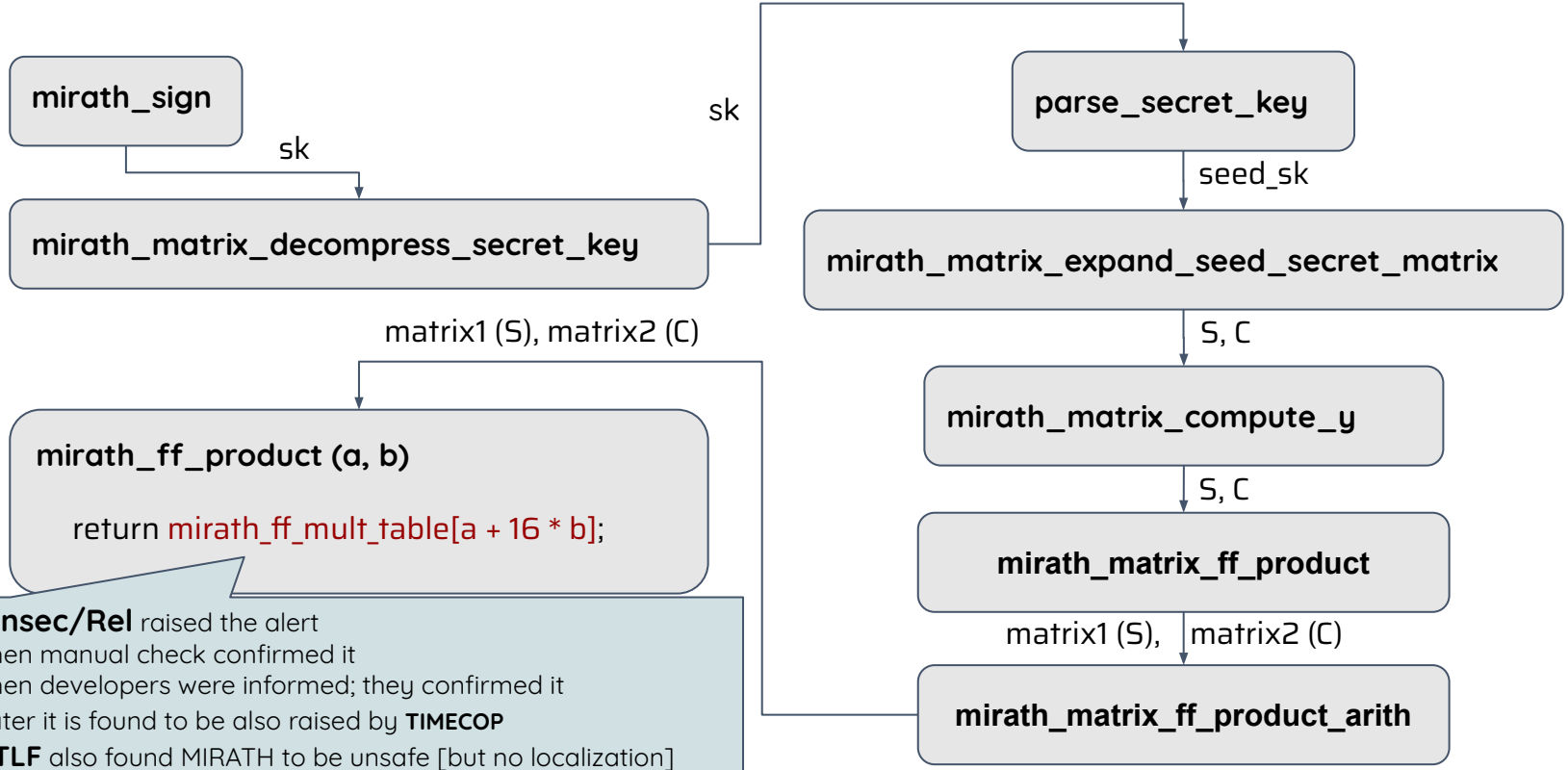
Potential existence of timing side-channel

Critical: confirmed by developers to be fixed
Non-critical: CT-violation considered not critical by developers
Secure*: no violation found

What Can Go Wrong?

- Imprecise analysis
  - Ex: taint approximation in TIMECOP
  - The alert is not a feasible CT violation
- Improper secret specification
  - Not always so easy to understand what should be secret
- CT violations are not always considered exploitable

Case Study: Exploitation Discovery (MIRATH)



Case Study: Man vs Machine (in Ryde)

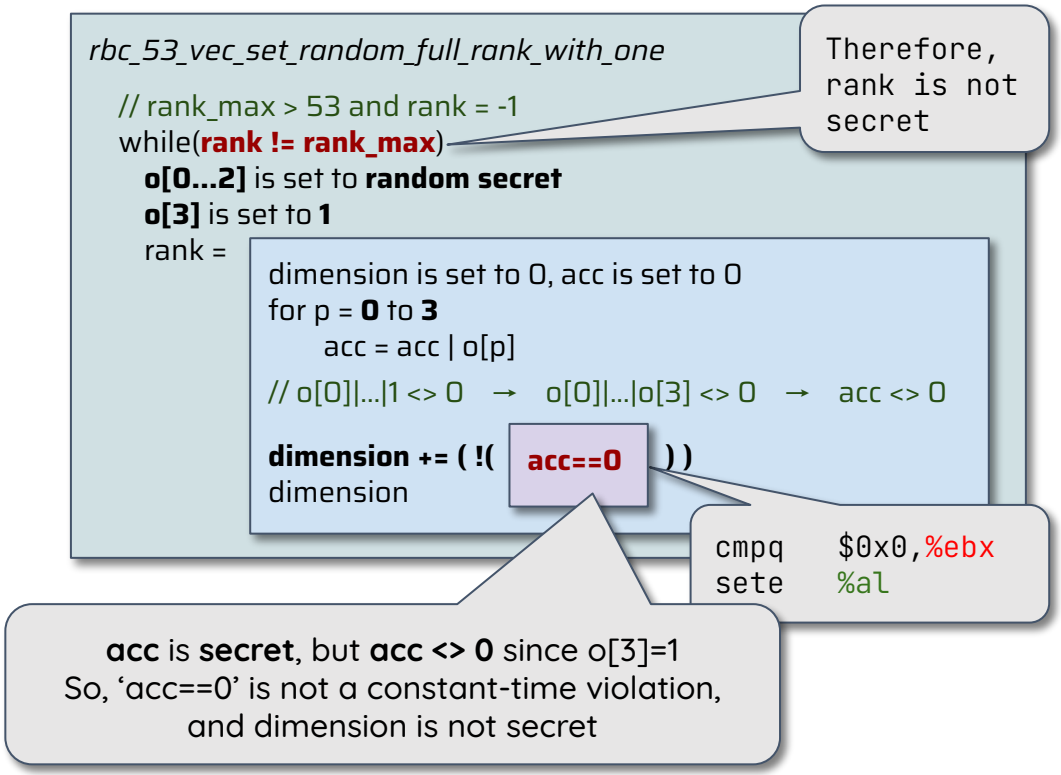
Timecop raised an alert at **rbc_53_vec_set_random_full_rank_with_one** (rbc_53_vec.c:109)

Initial checking revealed another leak and developers were informed

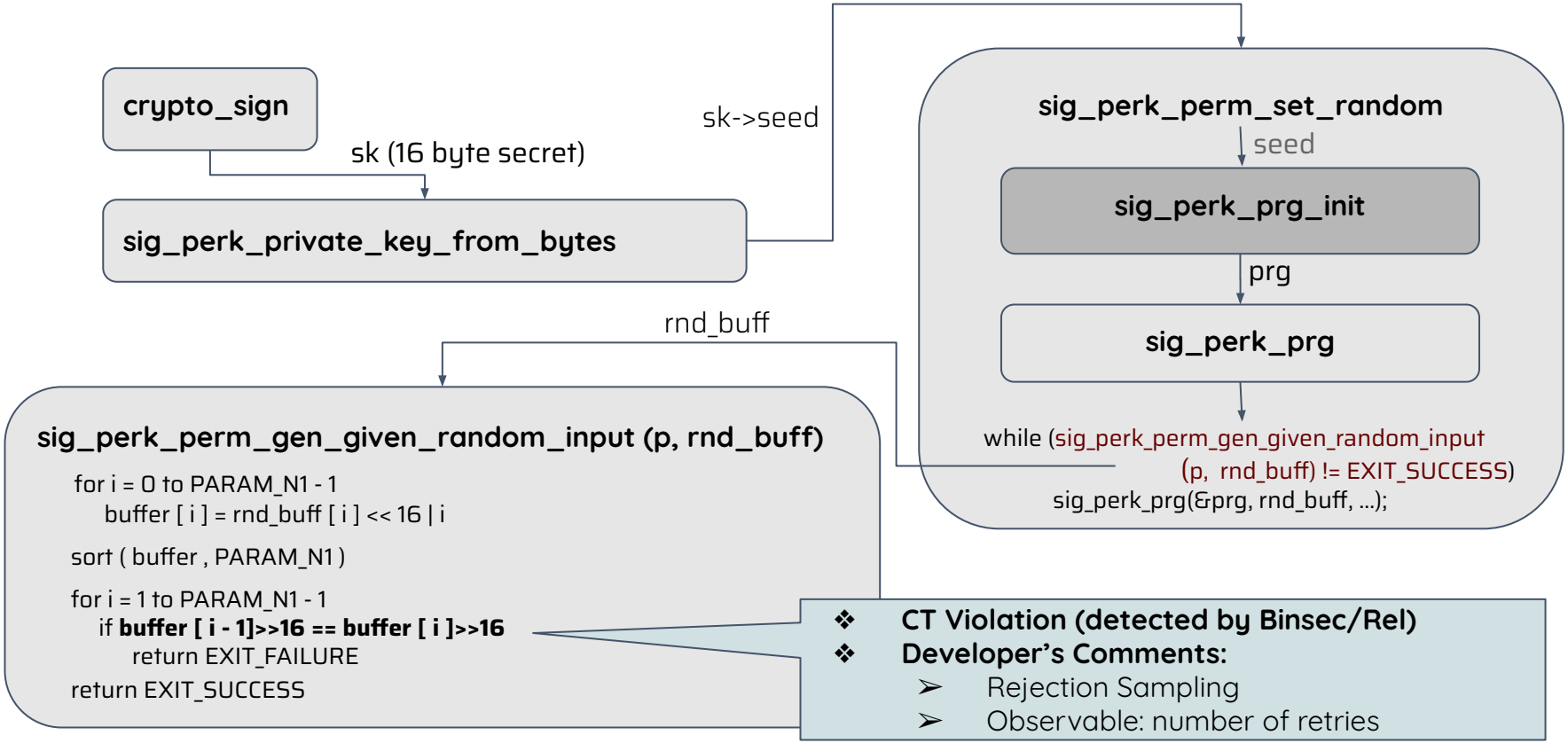
Developers agreed it to be a true constant-time violation

Binsec/Rel mysteriously did not raise the same alert

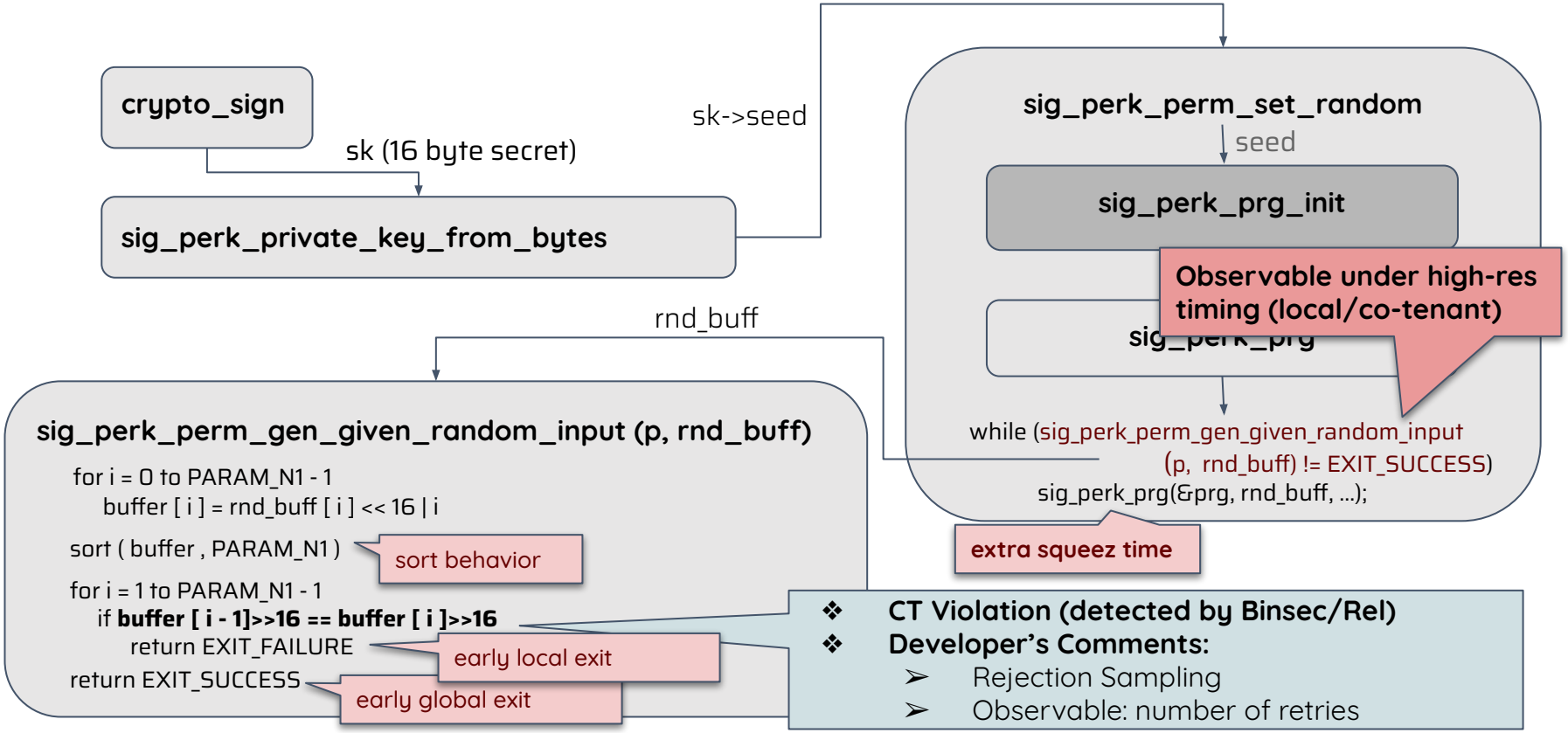
A curious re-investigation reveals it to be a false positive and **Binsec/Rel** to be correct



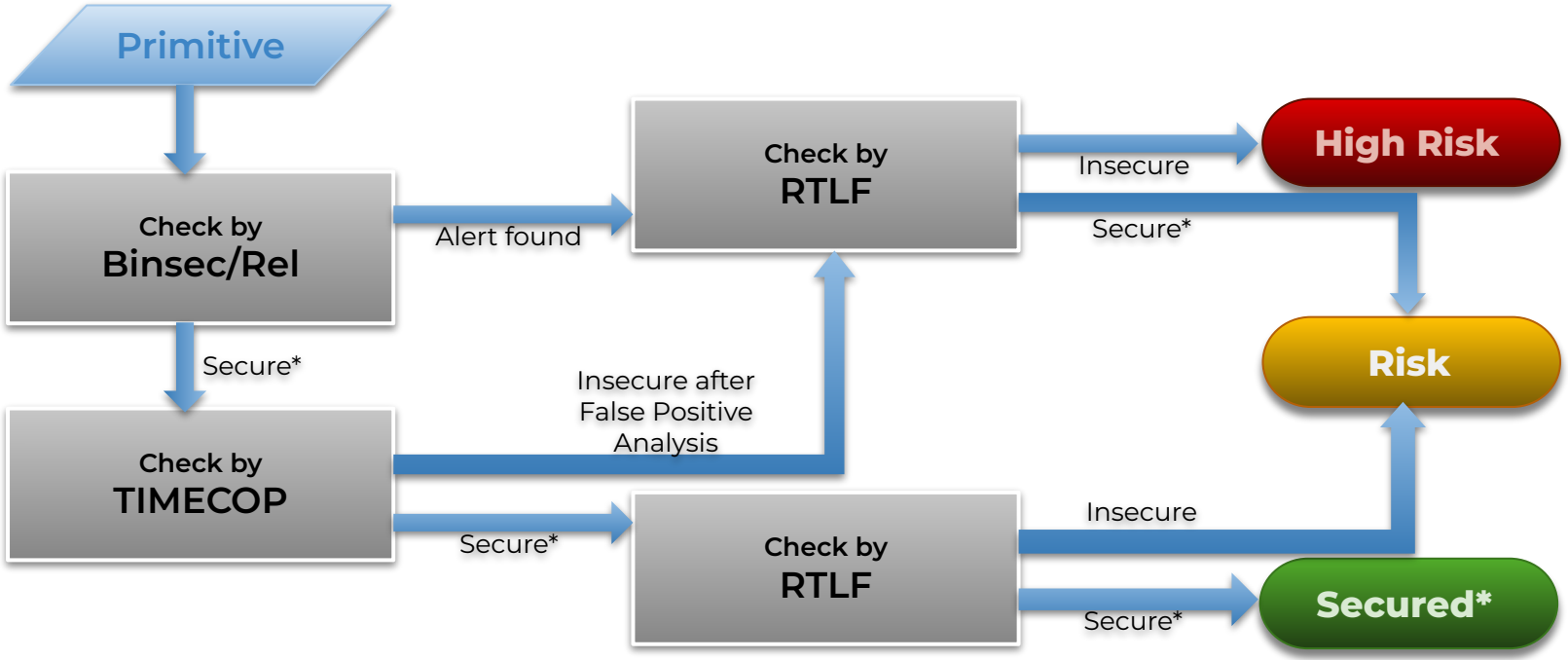
Case Study: Non-Critical vs. CT-V (PERK)



Case Study: Non-Critical vs. CT-V (PERK)



Recommended Process

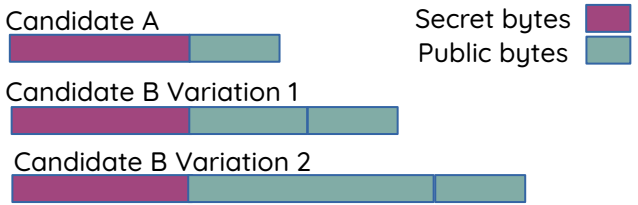


What Would Have Helped

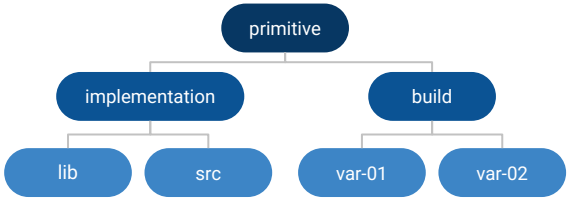
Standards in build parameters and compilation flags for all primitives

Ex. -O3 -g3 -static

Standards in placement of secret bytes in the secret keys



Standard Directory Structures



Use of common test harness is crucial

Conclusion

22 violations reported

5 fixes in 4 implementations

58 alerts still to process

Our toolchain

- is built to assist PQC standardization process by NIST
- integrates major PQC primitives
- supports analysis of generic programs as well
- leverages mutual strength of various automated tools

Our experiments

- uncovers huge amount of constant-time violations
- valid violations have been reported to the developers
- exploitable codes are fixed by the developers

Take Away

Step 1. use our automatic evaluation toolkit
Step 2. address first the alerts of **Binsec/Rel** for its trustiness and rigor analysis
Step 3. address the rest of alerts (by TIMECOP) for more coverage
Step 4. take **RTLF**'s analysis to assess the severity of the side-channels

What is Necessary

- standards in build configuration and compilation flags for all primitives
- standards in placement of secret bytes in the secret keys
- standards in common directory structures
- use of common test harness is crucial


Thank you very much everyone