STAnalyzer: A Simple Static Analysis Tool for Detecting Cache-Timing Leakages

Alexander Schaub, Olivier Rioul & Sylvain Guilley

June 24, 2019

Layout of the Presentation

Layout of the Presentation

OS Memory Model

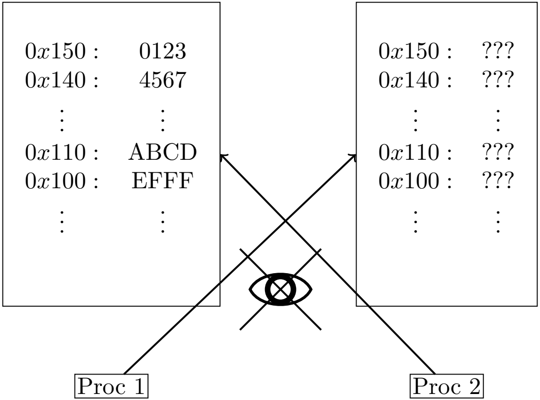


Figure: Per-process memory isolation.

Memory Sharing

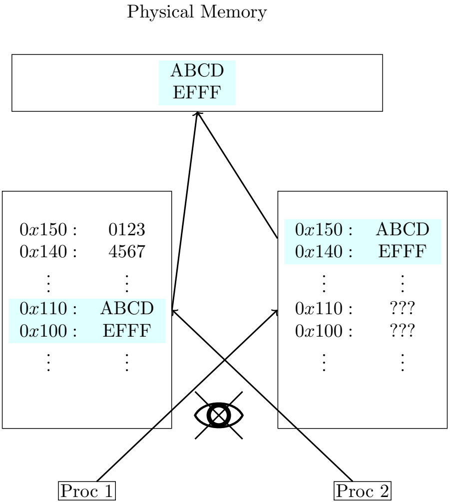


Figure: Shared memory (dynamically-linked libraries, page duplication,...)

Cache-Line Sharing

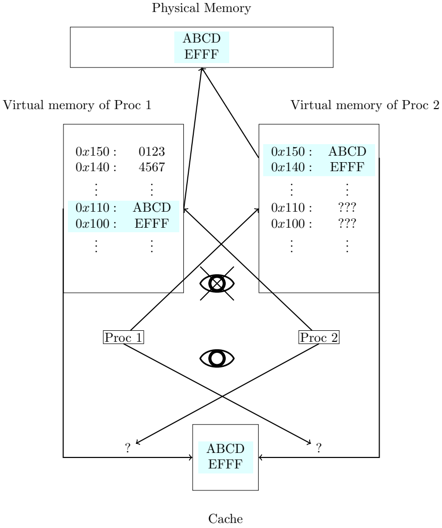


Figure: Cache-line sharing between processes.

How to Determine the Presence of Data in the Cache ?

Several techniques exist, for instance:

- PRIME + PROBE[1,2]
- EVICT + TIME [3]
- FLUSH + RELOAD[3]

Example to follow...

---

[1] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES", , in *Cryptographers Track at the RSA Conference*, Springer, 2006, pp. 1–20.

[2] F. Liu, Y. Yarom, Q. Ge, *et al.*, "Last-level cache side-channel attacks are practical", in *Security and Privacy (SP), 2015 IEEE Symposium on*, IEEE, 2015, pp. 605–622.

[3] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack.", in *USENIX Security Symposium*, 2014, pp. 719–732.

Example: FLUSH+RELOAD

Attacker	Victim	Remark
`clflush` *addr*		*addr* absent from cache
	executes code	*addr* might be present
`a = rdtsc()`		
`load` *addr*		if the load was fast, the attacker now knows that *addr* was accessed
`store rdtsc() - a`		
`clflush` *addr*		*addr* absent from cache
	executes code	
...		

Recognizing Vulnerable Code

What / How	Data	Code
Exploit	Sensitive indirections	Conditional jump/call
Reason	Memory load	Code execution
Code vulnerability	Derefencing a pointer to a secret-dependent address	Branching on a secret-dependent condition

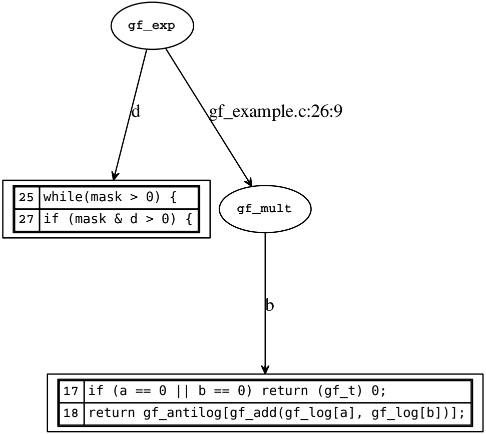
Note: FLUSH + RELOAD only applicable to **shared** data or code (static arrays, code in shared dynamic libraries, etc.)

Vulnerable Code

```c
static gf_t[] gf_antilog = {...};
static gf_t[] gf_log = {...};

gf_t gf_mult(gf_t a, gf_t b) {
  if (a == 0 || b == 0) return 0;
  return gf_antilog[
  gf_add(gf_log[a], gf_log[b])];
}

gf_t gf_exp(gf_t b, unsigned d) {
  gf_t r = gf_one();
  mask = 1<<floor(log2(d));
  while(mask > 0) {
    r = gf_mult(r, r);
    if (mask & d > 0) {
          r = gf_mult(a, r);
    }
    mask /= 2;
  }
}
```



Layout of the Presentation

Problem Definition

- Given a **C program**, with **annotations** corresponding to **sensitive variables**, determine whether the program is potentially vulnerable to **cache-timing** side channel leaks.
- Solution should be **easy to use**, as **accurate** as possible, and applicable to **most cryptographic implementations** written in C.

General Approach

- General idea: perform value dependency propagation, and record table accesses / branching operations depending on sensitive data.
- Values tracked for dependency analysis are sensitive values and initial values of function arguments
- Algorithm consist in tracking the state of three objects during the exploration of the AST:
  - Dependencies between variables and values, as a bipartite graph G
  - List of leaking variables, with corresponding code instruction, call graph and dependency chain, L
  - "Additional" dependencies, to take branching behavior into account, as a set of values I

Semantics for Simple Operations

inst	$G' = \phi_G(G, I; \text{inst})$	$L' = \phi_L(L, G; \text{inst})$	I'
var $=$ expr	$G \sqcup \{\text{var} \to G(\langle \text{expr} \rangle) \cup I\}$	L	I
var op$_2$$=$ expr	$G \cup \{\text{var} \to G(\langle \text{expr} \rangle) \cup I\}$	L	I
var[expr$_1$] $=$ expr$_2$	$G \cup \{* \, \text{var} \to G(\langle \text{expr}_2 \rangle) \cup I\}$	$L \cup G(\langle \text{expr}_1 \rangle)$	I
if(expr){inst}	$\phi_G(G, I'; \text{inst})$	$G(\langle \text{expr} \rangle) \cup \phi_L(L, G; \text{inst})$	$I \cup G(\langle \text{expr} \rangle)$
return expr	$G \cup \{\backslash \text{RET} \to G(\langle \text{expr} \rangle) \cup I\}$	L	I

Note: analyzing loops consists in computing a fixed point, and a function call in applying a previously determined dependency graph, after translating variable names.

Pointer Handling

- C pointers make the value analysis more complicated - values can be aliased, for instance
- Solution: for each pointer, build a set of memory locations it *might* point-to
- On every pointer assignment, update this set according to the set of the assignee.
- Formalized by Andersen[4], known as "points-to" analysis.
- Might overestimate the set of possible memory locations, but this is necessary in order to avoid false positives.

[4] L. O. Andersen, *Program analysis and specialization for the C programming language*, 1994.

Pointer Handling Example

```
void foo(int a) {
        int *p = malloc(8); // &p: {p}
        int *q = malloc(8);  // &p: {p}, &q: {q}

        if (a > 0) {
                q = p; // &p: {p}, &q: {p}
        }
        else {
                p = q; // &p: {q}, &p: {p}
        }
        // &p: {p, q}, &q: {p, q}

        ...
}
```


TELECOM Paris

Limitations

- Recursive functions not supported
- Complex goto operations not supported (but fixable)
- Casts between different structures, or between different pointer indirections are not correctly handled, e.g. `*(int **)p` when chasing pointers
- Incorrect or "risky" code could in theory lead to missed leakages, because of buffer overflows, array out-of-bound accesses, or obfuscated pointer arithmetic.

False Positives

False positives can arise in some situations, for instance when:

- the result of an operation involving sensitive values, is not sensitive itself (the value of `s-s` does not depend on `s`, or the hash of a sensitive value might not be sensitive)
- dead code is into account, e.g. `if (condition_that_never_happens) { leak_sensitive_value(s);}` will still count as a leakage
- conditional code is turned into constant-time code by the compiler

TELECOM Paris

Layout of the Presentation

NIST Post-Quantum Cryptography Contest - Overview

- Quantum computers will break asymmetric cryptography
- Alternatives to RSA and ECC need to be developed and vetted for security, evaluated for performance
- 69 algorithms submitted to NIST, mostly lattice-based, code-based and multivariate cryptography
- Selection for the second round announced in January 2019

Results

Vulnerable Implementations

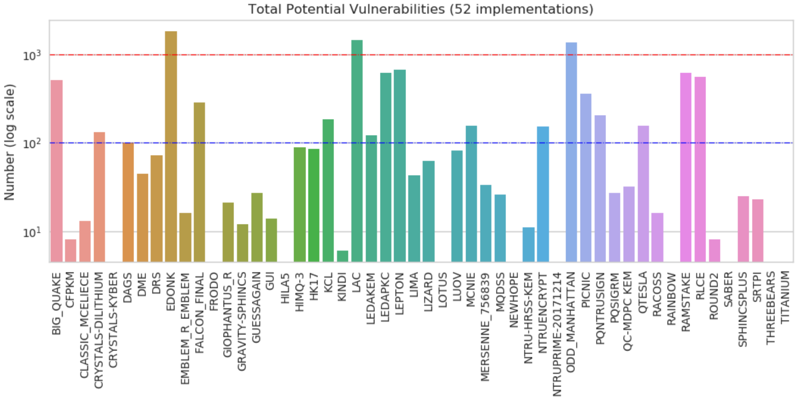


Figure: Total number of potential vulnerabilities found for each analyzed candidate

Note: 52 out of the 69 submissions were analyzed.

Results

Vulnerable Implementations

Out of 52 analyzed candidates:

- Potential vulnerabilities in **42** submissions (80.8%)
  - More than 100 reported vulnerabilities in **17** submissions
  - More than 1000 reported vulnerabilities in **3** submissions
- 4 submissions with easily fixable / probably not exploitable vulnerabilites (EMBLEM, Lima, Giophantus, OKCN-AKCN in the MLWE variant)
- 10 Submissions without detected vulnerabilites (Frodo, Rainbow, Hila5, Saber, CRYSTALS-Kyber, LOTUS, NewHope, ntruprime, ThreeBears and Titanium)

Results

Types of Vulnerabilities

We noticed some repeating patterns in the detected vulnerabilities.

- Gaussian sampling leak
- Other sampling leaks
- GMP library use (at least the standalone implementation)
- Operations in finite fields
- Other: AES re-implementation, matrix operations, error-decoding

  ...

TELECOM Paris

Layout of the Presentation

Cache-Timing Attacks
- Introduction
- Example Vulnerable Code

Static Code-Analysis
- Problem Statement
- Semantics
- Limitations

Results
- Analysis of First Round NIST PQC Standardization Candidates

Conclusion

Conclusion

- We presented STAnalyzer, an algorithm and a tool to detect potential side-channel leakages in C implementations
- Our program is able to analyze even large, unmodified programs, as shown by our analysis of most post-quantum proposals submitted to NIST
- There are no missed leaks with this approach, at the cost of a few false positives
- Not all leakages are exploitable, but assessing their exploitability automatically is a hard problem.
- **Perspective**: combining static analysis techniques with a dynamic analysis could allow us to assess the exploitability of the detected vulnerabilities and provide more information of practical importance.