Fine-Grained Constant-Time Policies with Jasmin & EasyCrypt

Vincent Laporte;
and the Formosa Crypto team members

2022-11-24, GLSec, CNAM, Paris

Remote Side-Channel Attacks

1996: Timing attacks on implementations…, Paul Kocher
2003: Remote timing attacks are practical, Brumley & Boney
2005: Cache missing for fun and profit; Cache-timing attacks against AES
2010: Efficient cache attacks on AES
2011: Cache Games
2013: Lucky Thirteen
2014: Flush+Reload
2015: Subnormal floating point; Lucky microseconds
2016: Cache bleed; Armageddon
2017: May the 4th be with you
2018: Spectre
2022: Hertz bleed

Constant-Time Programming

Two (basic) simple rules

- no branching on secret data
- no memory access at secret addresses

Pros

- Simple
- Source-level counter-measure

Cons

- Unusual coding patterns
- May negatively impact performances

Assembly-Level Security

Constant-Time programming is a **source-level** countermeasure. . .

Compilers are too eager at optimizing

- b $\times$ p $+$ $(1 - b)$ $\times$ q becomes if b then p else q
- run-time libraries have fast path for small operands

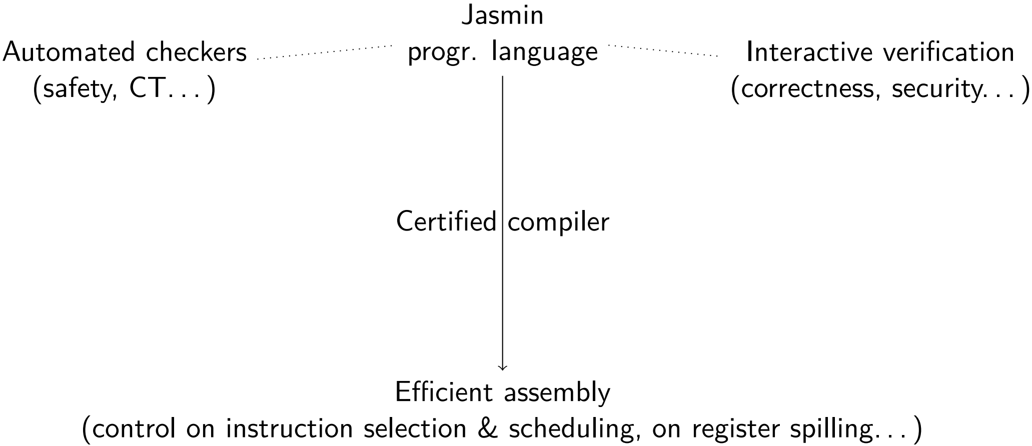
Verifying CT Security

For each branching condition and each dereferenced address prove that it is public.

- Simple **taint** analysis
- Needs prior knowledge about which inputs are public

Main difficulty: pointer analysis (aka alias analysis)

Jasmin Workbench



Jasmin
progr. language

Automated checkers
(safety, CT. . .)

Interactive verification
(correctness, security. . .)

Certified compiler

Efficient assembly
(control on instruction selection & scheduling, on register spilling. . .)

Jasmin Example

```
export
fn lehmer(reg u64 state) ⟶ reg u64 {
    reg u64[2] s, m;
    stack u64[2] t;
    inline int i;
    reg u64 j, result;
    for i = 0 to 2 {
        s[i] = [state + i * 8];
    }
    m[0] = 0x261fd0407a968add;
    m[1] = 0x45a31efc5a35d971;
    t = mul128(s, m);
    result = t[1];
    j = 0;
    while (j < 2) {
        [state + j * 8] = t[(int) j];
        j += 1;
    }
    return result;
}
```

```
inline
fn mul128(reg u64[2] x, y) ⟶ stack u64[2] {
    reg u64 xhi, ylo, lo, hi, tmp;
    stack u64[2] r;
    xhi = x[1];
    ylo = y[0];
    hi, lo = #MULX(ylo, x[0]);
    tmp = xhi * y[0];
    hi += tmp;
    y[1] *= x[0];
    y[1] += hi;
    r[0] = lo;
    r[1] = y[1];
    return r;
}
```

Constant-Time, Formally

Instrumented Semantics

The adversary observes:

- control flow
- memory accesses:

$$f : (\vec{a}, m) \Downarrow_{p}^{\ell} (\vec{r}, m')$$

Syntax of Structured Leakage

$$\begin{array}{rcll@{\qquad}rcll}
\ell_e & ::= & \bullet & \text{empty} & \ell & ::= & \ell_e := \ell_e & \text{assignment} \\
 & \mid & v & \text{value} & & \mid & \mathsf{if}_b(\ell_e, \ell) & \text{conditional} \\
 & \mid & (\ell_e, \ldots, \ell_e) & \text{sub-leakage} & & \mid & \mathsf{while}_{\mathsf{t}}(\ell_e, \ell, \ell) & \text{iteration} \\
 & & & & & \mid & \mathsf{while}_{\mathsf{f}}(\ell_e) & \text{loop end} \\
 & & & & & \mid & \{\ell; \ldots; \ell\} & \text{sequence}
\end{array}$$

Security property (φ-CT)

Given a “low-equivalence” relation φ between initial states:

$$\forall \vec{a}_1 \; m_1 \; \vec{a}_2 \; m_2 \; \ell_1 \; \ell_2 \; \vec{r}_1 \; \vec{r}_2 \; m_1' \; m_2',$$

$$(\vec{a}_1, m_1) \varphi (\vec{a}_2, m_2) \implies \begin{cases} f : (\vec{a}_1, m_1) \Downarrow_{p}^{\ell_1} (\vec{r}_1, m_1') \\ f : (\vec{a}_2, m_2) \Downarrow_{p}^{\ell_2} (\vec{r}_2, m_2') \end{cases} \implies \ell_1 = \ell_2.$$

Example: Explicit Leakage

```
fn max(reg u64[2] a) ⟶ reg u64 {
    reg u64 x, y;
    x = a[0];
    y = a[1];
    if x < y {
        x = y;
    }
    return x;
}
```

Applied to { 19; 23 }, it yields:

$$\left\{ \begin{array}{l} \bullet := (\bullet, 0); \\ \bullet := (\bullet, 1); \\ \mathsf{if}_{tt}(((\bullet, \bullet), \bullet), \{\bullet := \bullet\}) \end{array} \right\}$$

Quizz

Is this function constant-time?

A Type-System for Constant-Time

- Two-level security lattice: $\mathsf{L} \leqslant \mathsf{H}$
- A (flow-sensitive) typing environment mapping variables to security level
- Usual type-system for non-interference with public branching only (no implicit flows)
- Selected rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash x := e : \Gamma\{x \leftarrow \tau\}}\textsc{Assign}$$

$$\frac{\Gamma \vdash i : \Gamma_i \quad \Gamma_i \vdash c : \Gamma'}{\Gamma \vdash i; c : \Gamma'}\textsc{Seq} \qquad \frac{\Gamma \vdash b : \mathsf{L} \quad \Gamma \vdash c_1 : \Gamma_1 \quad \Gamma \vdash c_2 : \Gamma_2}{\Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 : \Gamma_1 \sqcup \Gamma_2}\textsc{Cond}$$

$$\frac{\Gamma \vdash p : \mathsf{L} \quad \Gamma \vdash e : \tau}{\Gamma \vdash *p := e : \Gamma}\textsc{Store} \qquad \frac{\Gamma \vdash n : \mathsf{L} \quad \Gamma \vdash e : \tau}{\Gamma \vdash a[n] := e : \Gamma\{a \leftarrow \tau \sqcup \Gamma(a)\}}\textsc{Assign-Array}$$

A Wide Range of Adversary Models

The *base-line* adversary model does not fit all scenarios: sometimes too coarse, sometimes too precise.

Time-Variable instructions

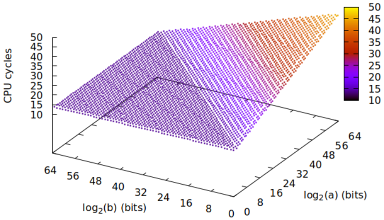


Figure 1: Timing behavior of the 64-bit div instruction on a x86 microprocessor

- Need more precise leakage model

Caches operate at a *line* level

Can observers *really* distinguish addresses within a single cache-line? Defenses under such threat model are expensive.

- Need less precise leakage model

Leakage models form a **lattice** (A. Shivakumar, Barthe, Grégoire, Laporte, and Priya 2022)

Interactive Proofs with EasyCrypt

- A program translation $[\cdot]$ to EasyCrypt makes leakage explicit
- Security (program c is φ-CT) is reduced to a **Relational Hoare Logic** statement:

$$[c] \sim [c] : \varphi \wedge =_{\{\mathsf{leak}\}} \implies =_{\{\mathsf{leak}\}}$$

- Reasoning is vastly simplified through the use of automated tactics (wp, smt. . .)
- Two parameters model:
  - leakage of operators
  - leakage of memory accesses

```
clone import ALeakageModel as LeakageModel.

module M = {
  var leak : leakages_t

  proc max (a: W64.t Array2.t) : W64.t = {
    var aux: W64.t;

    var x : W64.t;
    var y : W64.t;

    leak <- LeakAddr([0]) :: leak;
    aux <- a.[0];
    x <- aux;
    leak <- LeakAddr([1]) :: leak;
    aux <- a.[1];
    y <- aux;
    leak <- LeakCond(x \ult y) :: LeakAddr([]) :: leak;
    if (x \ult y) {
      aux <- y;
      x <- aux;
    }
    return (x);
  }
}.
```

Application: Efficient Constant-Time Modulo

```
export fn mod_TV(reg u64 a, reg u64 b) ⟶ reg u64 {
    reg u64 flag one zero dividend modulo result . . . ;
    reg bool lzaz lzbz cf;
    flag = 0x1234; one = 1; zero = 0;
    lzbz, lzb = lzcnt(b);
    flag = zero if lzbz;
    lzaz, _ = lzcnt(a);
    flag = one if lzaz;
    lzb_m1 = #LEA(lzb - 1);
    b_lzb_m1 = b << lzb_m1;
    dividend = #LEA(b_lzb_m1 + a);
    b_lzb = b_lzb_m1 << 1;
    cf, temp2 = b_lzb + a;
    dividend = temp2 if ! cf;
    dividend = a if flag == 1;
    temp2 = 0xFFFFFFFFFFFFFFFF;
    dividend = temp2 if flag == 0;
    modulo = dividend % b ;
    result = modulo;
    result = a if flag == 0;
    return result;
}
```

- Efficiently uses hardware division
- Timing no longer depends on the value of the first argument (a):

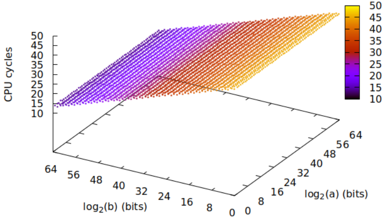


Figure 2: Timing behavior of mod_TV

- Proved **correct** and **secure** in EasyCrypt

Compiler Correctness

Semantics Preservation (forward simulation)

If the compilation of program p produces a program p',
then its safe behaviors are preserved:

$$\forall \vec{a}\ m\ \vec{r}\ m', \qquad f : (\vec{a}, m) \Downarrow_p (\vec{r}, m') \qquad \Longrightarrow \qquad f : (\vec{a}, m) \Downarrow_{p'} (\vec{r}, m').$$

Hidden Details

- Source and target languages are different
- Initial states are not the same (but tightly related)
- The target stack must be large enough
  - i.e., the compiler does not enforce the absence of “stack overflow”

Compilation & Instrumented Semantics

Compilers do not preserve leakage

- common subexpression elimination removes observable events
- loop invariant code motion reorders events
- allocation of local variables introduces events
- instruction selection transforms events
- . . .

Is this constant-time preserving?

Instrumented Correctness

Leakage Transformers

If the compilation of program p produces a program p',
there exists a leakage transformer F such that:

$$\forall \vec{a}\ m\ \ell\ \vec{r}\ m', \qquad f : (\vec{a}, m) \Downarrow_{p}^{\ell} (\vec{r}, m') \qquad \Longrightarrow \qquad f : (\vec{a}, m) \Downarrow_{p'}^{F(\ell)} (\vec{r}, m').$$

Corollary

Constant-Time security is preserved

Beyond CT-Preservation

Leakage transformers allow to reason about **target instrumentation** at the source level (e.g., execution time, resource usage. . .)
See recent work, focusing on execution cost (Barthe et al. 2021)

Branch Prediction and Spectre

```
u64[2] offsets = { 1, 0 };
u64[2] data = { 0xcafe, 0xbeef };

export
fn archetype(#public reg u64 n) ⟶ reg u64 {
    reg u64 p r;
    r = 0;
    if n < 2 { // Branch trained with in-bounds n
        // Speculatively, out-of-bounds
        p = offsets[(int) n];
        // Speculatively, load from secret address
        // → Cache attack!
        r = data[(int) p];
    }
    return r;
}
```

Security goal: speculative constant-time

No choice of speculation can make the leakage depend on sensitive information

Spoiler

Counter-measures can be efficiently implemented and automatically verified (A. Shivakumar, Barthe, Grégoire, Laporte, Oliveira, et al. 2022)

Open Questions

- Precisely check SCT at assembly-level?
- Justify that compilation preserves SCT?

Thanks

Get in touch at https://formosa-crypto.org

References

A. Shivakumar, Basavesh, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2022. "Typing High-Speed Cryptography Against Spectre V1." Cryptology ePrint Archive, Paper 2022/1270. https://eprint.iacr.org/2022/1270.

A. Shivakumar, Basavesh, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. "Enforcing Fine-grained Constant-time Policies." In *CCS '22: 2022 ACM SIGSAC Conference on Computer and Communications Security*, 83–96. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22). Los Angeles CA, United States: ACM. https://doi.org/10.1145/3548606.3560689.

Barthe, Gilles, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. "Structured Leakage and Applications to Cryptographic Constant-Time and Cost." In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 462–76. CCS '21. New York, NY, USA: ACM. https://doi.org/10.1145/3460120.3484761.