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
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
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

Automated checkers (safety, CT...)  →  Jasmin progr. language  →  Interactive verification (correctness, security...)

Certified compiler

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

$$f : (\vec{a}, m) \downarrow^l_p (\vec{r}, m')$$

### Syntax of Structured Leakage

$$\ell_e ::= \bullet \quad \text{empty} \quad \ell ::= \ell_e ::= \ell_e \quad \text{assignment}$$

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>empty statement</td>
</tr>
<tr>
<td>value</td>
<td>value assignment</td>
</tr>
<tr>
<td>sub-leakage</td>
<td>sub-leakage</td>
</tr>
<tr>
<td>if (b(\ell_e, \ell))</td>
<td>conditional branch</td>
</tr>
<tr>
<td>while_t(\ell_e, \ell, \ell)</td>
<td>iteration loop</td>
</tr>
<tr>
<td>while_f(\ell_e)</td>
<td>loop end</td>
</tr>
<tr>
<td>{\ell; \ldots; \ell}</td>
<td>sequence</td>
</tr>
</tbody>
</table>

### Security property (\(\varphi\)-CT)

Given a “low-equivalence” relation \(\varphi\) between initial states:

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

$$(\vec{a}_1, m_1)\varphi(\vec{a}_2, m_2) \implies \begin{cases} f : (\vec{a}_1, m_1) \downarrow^l_1 (\vec{r}_1, m'_1) \\ f : (\vec{a}_2, m_2) \downarrow^l_2 (\vec{r}_2, m'_2) \end{cases} \implies \ell_1 = \ell_2.$$
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:

\{ • := (•, 0);
  • := (•, 1);
  if \#(((•, •), •), {• := •})  \}

Quizz

Is this function constant-time?
A Type-System for Constant-Time

- Two-level security lattice: $L \leq H$
- 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:

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash x := e : \Gamma\{x \leftarrow \tau\}} \text{ ASSIGN}
\]

\[
\frac{\Gamma \vdash i : \Gamma_i \quad \Gamma_i \vdash c : \Gamma'}{\Gamma \vdash i; c : \Gamma'} \text{ SEQ}
\]

\[
\frac{\Gamma \vdash b : L \quad \Gamma \vdash c_1 : \Gamma_1 \quad \Gamma \vdash c_2 : \Gamma_2}{\Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 : \Gamma_1 \sqcup \Gamma_2} \text{ COND}
\]

\[
\frac{\Gamma \vdash p : L \quad \Gamma \vdash e : \tau}{\Gamma \vdash *p := e : \Gamma} \text{ STORE}
\]

\[
\frac{\Gamma \vdash n : L \quad \Gamma \vdash e : \tau}{\Gamma \vdash a[n] := e : \Gamma\{a \leftarrow \tau \sqcup \Gamma(a)\}} \text{ ASSIGN-ARRAY}
\]
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](image)

- 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 \(\varphi\)-CT) is reduced to a Relational Hoare Logic statement:

\[
\begin{align*}
[c] & \sim [c] : \varphi \land \{\text{leak}\} & \implies & \{\text{leak}\}
\end{align*}
\]

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

```plaintext
clon 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);
  }
}.```
Efficiently uses hardware division

Timing no longer depends on the value of the first argument \( (a) \):

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:

$$\forall \vec{a} \ m \ \vec{r} \ m', \ f : (\vec{a}, m) \downarrow_p (\vec{r}, m') \implies f : (\vec{a}, m) \downarrow_{p'} (\vec{r}, m').$$

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:

\[ \forall \vec{a} \ m \ \ell \ \vec{r} \ m', \quad f : (\vec{a}, m) \downarrow^\ell_p (\vec{r}, m') \implies f : (\vec{a}, m) \downarrow^{F(\ell)}_{p'} (\vec{r}, m'). \]

**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

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

