Type-directed Program Transformation for Constant-Time Enforcement

Gautier Raimondi

SCRATCHS

November 22, 2021

Timing attack

Timing attacks are based on analysing the time taken to execute algorithms.
Time leaks information on secrets if the behaviour of the program is dependant of the value of the secrets. For example, conditional jump or division secret value.

Proved practical in 2005 by Brumley and Boneh[1], by extracting private keys from an OpenSSL-based web server running on a machine in the local network

[1]D. Brumley and D. Boneh, "Remote timing attacks are practical" (2005)

A solution : Constant-Time programming

A programming policy to defend implementations against timing attacks.

Further constraints are imposed :

- No control flow decisions using secret data
- No memory access using secret data as index

Both hard to write, and maintain

How to obtain Constant-Time ?

- Assembly verification (Barthe, Betarte, et al. 2019)
- Source verification (Blazy, Pichardie, and Trieu 2019)
- Program transformation (SC-Eliminator, Wu et al. 2018)
- DSLs (VALE, JASMIN, FaCT ...)

JASMIN[2]

- As VALE, a DSL/compiler for writing low-level code.
- Unlike VALE, allow for higher-level construction.
- Prove constant-time by using the EASYCRYPT proof-assistant.

[2] José Bacelar Almeida et al., "Jasmin: High-Assurance and High-Speed Cryptography" (2017)

FaCT[3]

- A DSL to write constant-time code
- Rejects non-transformable programs
- Ensures that post-transformation, the program is indeed constant-time

[3]Sunjay Cauligi et al., "FaCT: a DSL for timing-sensitive computation" (2019)

Objectives

- Expand the class of programs transformable by FaCT
- Define a type-system to accept this class of programs
- Implement this transformation into Jasmin

Language

Simple imperative language with arrays and a **for** loop.

$$\begin{array}{rcl} expr \ni e & ::= & x \mid c \mid e \oplus e \mid e?e : e \mid t[e] \\ stmt \ni s & ::= & \textbf{skip} \mid x = e \mid t[e] = e \mid s; s \\ & & \mid \textbf{if } x@p \textbf{ then } s \textbf{ else } s \\ & & \mid \textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s \end{array}$$

$e_1?e_2{:}e_3$ is a constant-time conditional expression, implemented either by bitwise operations or `cmove`, and do not leak e_1.
$@p$ is used to identify each condition.
We also assume initial program is safe.

Leakage semantics

$$\frac{}{(c, \sigma) \downarrow^{\epsilon} c} \qquad \frac{}{(x, \sigma) \downarrow^{\epsilon} \sigma(x)}$$

$$\frac{(e_1, \sigma) \downarrow^{t_1} v_1 \quad (e_2, \sigma) \downarrow^{t_2} v_2 \quad v_3 = v_1 \oplus v_2}{(e_1 \oplus e_2, \sigma) \downarrow^{t_1 \cdot t_2} v_3}$$

$$\frac{\begin{array}{c}(e_i, \sigma) \downarrow^{t_i} v_i \quad i \in \{1, 2, 3\} \\ v = \mathit{if}\ v_1\ \mathit{then}\ v_2\ \mathit{else}\ v_3\end{array}}{(e_1 ? e_2 : e_3, \sigma) \downarrow^{t_1 \cdot t_2 \cdot t_3} v}$$

$$\frac{(e, \sigma) \downarrow^{t_e} i \quad 0 \leq i < \mathit{size}(t) \quad \sigma(t)[i] = v}{(t[e], \sigma) \downarrow^{t_e + i} v}$$

$$\frac{}{(\mathbf{skip}, \sigma) \downarrow^{\epsilon} \sigma} \qquad \frac{(e, \sigma) \downarrow^{t} v}{(x = e, \sigma) \downarrow^{t} \sigma[x \mapsto v]}$$

$$\frac{(s_1, \sigma_1) \downarrow^{t_1} \sigma_2 \quad (s_2, \sigma_2) \downarrow^{t_2} \sigma_3}{(s_1; s_2, \sigma_1) \downarrow^{t_1 \cdot t_2} \sigma_3}$$

$$\frac{(e_1, \sigma) \downarrow^{t_1} i \quad 0 \leq i < \mathit{size}(t) \quad (e_2, \sigma) \downarrow^{t_2} v}{(t[e_1] = e_2, \sigma) \downarrow^{t_1 \cdot t_2 \cdot i} \sigma[t \mapsto (\sigma(t)[i \mapsto v])]}$$

$$\frac{\sigma(x) = b \quad (s_b, \sigma) \downarrow^{t} \sigma'}{(\mathbf{if}\ x@p\ \mathbf{then}\ s_{\mathit{true}}\ \mathbf{else}\ s_{\mathit{false}}, \sigma) \downarrow^{b \cdot t} \sigma'}$$

$$\frac{\forall_{i \in [c_1; c_2]} (x = i; s, \sigma_i) \downarrow^{t_i} \sigma_{i+1} \quad t = t_{c_1} \cdot \dots \cdot t_{c_2}}{(\mathbf{for}\ e\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s, \sigma_{c_1}) \downarrow^{t} \sigma_{c_2+1}}$$

where σ is an environment, and $(P, \sigma) \downarrow^{t} \sigma'$ generates a leakage t.

Leakage is generated when evaluating a conditional or performing an array's access[4].

[4] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte, "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"" (2018)

Constant-Time

$$CT(p) \triangleq \bigwedge \begin{pmatrix} \sigma_1 \equiv_L \sigma_2 \\ (p, \sigma_1) \downarrow^{t_1} \sigma'_1 \\ (p, \sigma_2) \downarrow^{t_2} \sigma'_2 \end{pmatrix} \Rightarrow t_1 = t_2$$

where $\sigma \equiv_L \sigma' \triangleq \forall x \in L, \sigma(x) = \sigma'(x)$.

Constant-Time

$$CT(p) \triangleq \bigwedge \begin{pmatrix} \sigma_1 \equiv_L \sigma_2 \\ (p, \sigma_1) \downarrow^{t_1} \sigma_1' \\ (p, \sigma_2) \downarrow^{t_2} \sigma_2' \end{pmatrix} \Rightarrow t_1 = t_2$$

where $\sigma \equiv_L \sigma' \triangleq \forall x \in L, \sigma(x) = \sigma'(x)$.

$$\frac{\Gamma \vdash e : L}{\Gamma \vdash t[e] : \Gamma(t)} \qquad \frac{\Gamma \vdash e_1 : L \quad \Gamma \vdash t : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma\{t[e_1]{:=}e_2\}\Gamma[t \to \tau_1 \sqcup \tau_2]}$$

$$\frac{\Gamma \vdash x : L \quad \Gamma\{s_1\}\Gamma' \quad \Gamma\{s_2\}\Gamma'}{\Gamma\{\texttt{if } x@p \texttt{ then } s_1 \texttt{ else } s_2\}\Gamma'}$$

Type system

- Flow-insensitive type system
- Allows conditionals to branch on secret
- Disallows memory accesses in the scope of a secret conditional

If-conversion

Generates branchless and predicated code.

$$\textbf{if } s@p \textbf{ then } x = e_1 \textbf{ else } x = e_2$$

If-conversion

Generates branchless and predicated code.

$$\textbf{if } s@p \textbf{ then } x = e_1 \textbf{ else } x = e_2 \ \rightarrow x = s?e_1{:}x; x = !s?e_2{:}x$$

Turns indirect flows into direct flows.

Public safety

A program is *public safe* iff its safety can be proved using only publicly available information.

Public safety

A program is *public safe* iff its safety can be proved using only publicly available information.

$$\textbf{if } (i > s)@p \textbf{ then } x = t[i] \textbf{ else skip} \;\rightarrow x = i > s?t[i]{:}x \quad \times$$

Public safety

A program is *public safe* iff its safety can be proved using only publicly available information.

$$\textbf{if } (i > s)@p \textbf{ then } x = t[i] \textbf{ else skip } \rightarrow x = i > s?t[i]{:}x \quad \times$$

$$\textbf{if } s@p \textbf{ then } x = t[i] \textbf{ else skip } \rightarrow x = s?t[i]{:}x \quad ✔$$

The type system generates verification to ensures that the memory accesses are valid.

Delayed if-conversion I

if-conversion is not strong enough to handle leakage within a conditional.

$$\textbf{if } s@p \textbf{ then } x = l; y = t[x] \textbf{ else skip}$$

would yield

$$x = s?l{:}x; y = s?t[x]{:}y$$

which is insecure

Delayed if-conversion II

Postpones the *if-conversion* and direct flows.

$$\textbf{if } s@p \textbf{ then } x = l; y = t[x] \textbf{ else skip}$$

$$x_t = l; y_t = t[x_t]; x = s?x_t{:}x; y = s?y_t{:}y.$$

Out-of-scope

The information about indirect flow is not remembered long enough to handle *outside of conditions* array accesses.

$$(\textbf{if } s@p \textbf{ then } x = l_1 \textbf{ else } x = l_2\ ); y = t[x]$$

Out-of-scope

The information about indirect flow is not remembered long enough to handle *outside of conditions* array accesses.

$$(\textbf{if } s@p \textbf{ then } x = l_1 \textbf{ else } x = l_2\ ); y = t[x]$$

We move the access within the scope of the condition

$$\textbf{if } s@p \textbf{ then } x = l_1; y = t[x] \textbf{ else } x = l_2; y = t[x]$$

and after *if-conversion*

$$\begin{array}{l} x_t = l_1; y_t = t[x_t]; \\ x_e = l_2; y_e = t[x_e]; \\ x = s?x_t:x_e; \\ y = s?y_t:y_e; \end{array}$$

Out-of-scope - Upgraded I

Considering the snippet

$$\textbf{if } s@p \textbf{ then } x = 0 \textbf{ else } x = 1 \; ; t[y] = 0; t[t[y]] = 0; t'[x] = 1;$$

Not constant-time, because x is private.

Out-of-scope - Upgraded I

Considering the snippet

$$\textbf{if } s@p \textbf{ then } x = 0 \textbf{ else } x = 1 \; ; t[y] = 0; t[t[y]] = 0; t'[x] = 1;$$

Not constant-time, because x is private. However, we would obtain

$$x_1 = 1; t[y] = s?0{:}t[y]; t[t[y]] = s?0{:}t[t[y]]; \ldots ;$$

Not constant-time, independently of x, because t[y] is now secret.

Out-of-scope - Upgraded II

Problems :

- We don't remember which variables were public before moving them up a scope.
- The two branches are not synchronised, and a public variable in s_1, could not be in s_2.

Out-of-scope - Upgraded II

Problems :

- We don't remember which variables were public before moving them up a scope.
- The two branches are not synchronised, and a public variable in s_1, could not be in s_2.

Solution :

- Instead of only moving instructions up a scope, we save the continuation of the condition s_3 in a new construction :

$$\textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2 \quad \textbf{next } s_3$$

Safety is public

- The input program is safe
- Adding redundant checks doesn't affect semantics

∴ We can add redundant checks to memory accesses.

$$t[x] = e \to t[0 \leq x < n ? x : 0] = 0 \leq x < n ? e : t[0]$$

where n is the size of t
Size of arrays is public, so we don't introduce leakage.

Overview

- Flow-sensitive.
- Ignores public safety.
- Introduces leakage on direct-flow only.
- Extends the usual $\{\mathbf{H}, \mathbf{L}\}$ lattice with $\mathbf{I}(s)$

Type system

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash i : \mathbf{L}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \oplus e_2 : \tau_1 \sqcup \tau_2} \qquad \frac{\Gamma \vdash e_i : \tau_i \quad i \in \{1,2,3\}}{\Gamma \vdash e_1 ? e_2 : e_3 : \tau_1 \sqcup \tau_2 \sqcup \tau_3}$$

$$\frac{\Gamma \vdash t : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubset \mathbf{H}}{\Gamma \vdash t[e_2] : \tau_1 \sqcup \tau_2}$$

$$\frac{\Gamma \vdash e : \tau}{\kappa \vdash \Gamma\{x = e\}\Gamma[x \mapsto \tau \sqcup \kappa]} \qquad \frac{\Gamma \vdash t : \tau_t \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sqsubset \mathbf{H}}{\kappa \vdash \Gamma\{t[e_1] = e_2\}\Gamma[t \mapsto \tau_t \sqcup \tau_1 \sqcup \tau_2 \sqcup \kappa \ltimes \mathbf{H}]}$$

$$\frac{\kappa \vdash \Gamma\{p_1\}\Gamma_1 \quad \kappa \vdash \Gamma_1\{p_2\}\Gamma_2}{\kappa \vdash \Gamma\{p_1 ; p_2\}\Gamma_2} \qquad \frac{\begin{array}{c}\Gamma \vdash e : \tau \quad \kappa' = \kappa \sqcup (\tau \ltimes \mathbf{I}(\{p\})) \\ \kappa' \vdash \Gamma\{p_1\}\Gamma_1 \quad \kappa' \vdash \Gamma\{p_2\}\Gamma_2 \\ \Gamma_1 \sqsubseteq \Gamma' \quad \Gamma_2 \sqsubseteq \Gamma'\end{array}}{\kappa \vdash \Gamma\{\text{if } e_{@p} \text{ then } p_1 \text{ else } p_2\}\Gamma'} \qquad \frac{\Gamma \sqsubseteq \Gamma' \quad \kappa, \Gamma'[x \mapsto \kappa] \vdash s : \Gamma'}{\kappa \vdash \Gamma\{\textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s\}\Gamma'}$$

$$\text{where } \tau_1 \sqcup \tau_2 = \begin{cases} \mathbf{H} & \textit{if } \tau_1 = \mathbf{H} \vee \tau_2 = \mathbf{H} \\ \tau_1 & \textit{if } \tau_2 = \mathbf{L} \\ \tau_2 & \textit{if } \tau_1 = \mathbf{L} \\ \mathbf{I}(s_1 \cup s_2) & \textit{if } \tau_1 = \mathbf{I}(s_1) \wedge \tau_2 = \mathbf{I}(s_2) \end{cases} \qquad \tau \ltimes \tau' = \begin{cases} \mathbf{L} & \textit{if } \tau = \mathbf{L} \\ \tau' & \textit{otherwise} \end{cases}$$

Example

$$(\textbf{if } s@p \textbf{ then } x = l_1 \textbf{ else } x = l_2\ ); r = t[x]$$

with Γ such that

$$\Gamma \vdash s : \mathbf{H} \quad \Gamma \vdash e : \mathbf{L} \qquad e \in \{l_1, l_2, t\}.$$

Because s is secret, $x = l_1$ and $x = l_2$ are typed with a context $\kappa' = \mathbf{I}(\{p\})$
After the conditional, $\Gamma' = \Gamma[x \mapsto \mathbf{I}(\{p\})]$
At the end,

$$\Gamma'' = \Gamma'[r \mapsto \mathbf{I}(\{p\}).]$$

Overview

Three components :

- *if-selection*
- *index-sanitize*
- *branch-removal*

We cycle through these three components while there exists a secret condition in the program.

If-selection

- Locates a problematic $if_{@p}$ (using a secret guard).
- Transforms the condition to include its continuation (**next**)

The continuation is up to the last following instruction that is leaking a value of type $\mathbf{l}(s)$ with $p \in s$.

Public safety

Guarantees the safety of every memory access by adding dynamic checks.

$$\text{a-get}\ \frac{}{t[i] \rightarrow t[(0 \leq i < size(t))?i{:}0]}$$

$$\text{a-assgn}\ \frac{}{t[i] = e \rightarrow t[(0 \leq i < size(t))?i{:}0] = (0 \leq i < size(t))?e{:}t[0]}$$

N.B. This is done only on indexes codependant with the context.

Branch removal

We rename each branch, and delay the merge at the end of s_3

$$
\begin{array}{lcll}
[\![x = e]\!]^k_\rho & = & \rho(x) = [\![e]\!]_\rho & \\
[\![t[e_1] = e_2]\!]^k_\rho & = & t[[\![e_1]\!]_\rho] = k?[\![e_2]\!]_\rho : t[[\![e_1]\!]_\rho] & \\
[\![s_1; s_2]\!]^k_\rho & = & [\![s_1]\!]^k_\rho; [\![s_2]\!]^k_\rho & \\
[\![\textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s]\!]^k_\rho & = & \textbf{for } \rho(x) \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } [\![s]\!]^k_\rho & \\
[\![\textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ next } s_3]\!]^k_\rho & = & \left\{ \begin{array}{l} \textit{let } (\rho'_1, \rho'_2, s'_3) \textbf{ s.t } \rho_1, \rho_2, s_3 \rightarrow_{@p} \rho'_1, \rho'_2, s'_3 \\ \textbf{if } \rho(c) \textbf{ then } [\![s_1]\!]^k_\rho \textbf{ else } [\![s_2]\!]^k_\rho \textbf{ next } s'_3 \end{array} \right. & \text{if } \Gamma(c) = \mathbf{L} \\
[\![\textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ next } s_3]\!]^k_\rho & = & \left\{ \begin{array}{l} \textit{let } (\rho_1, \textit{pre}_1) = \textit{init}(\rho, s_1) \textit{in} \\ \textit{let } (\rho_2, \textit{pre}_2) = \textit{init}(\rho, s_2) \textit{in} \\ \textit{let } (\rho'_1, \rho'_2, s'_3) \textbf{ s.t } \rho_1, \rho_2, s_3 \rightarrow_{@p} \rho'_1, \rho'_2, s'_3 \\ \textit{pre}_1; \textit{pre}_2; [\![s_1]\!]^{k\&c}_{\rho_1}; [\![s_2]\!]^{k\&c}_{\rho_2}; s'_3 \end{array} \right. & \text{if } \Gamma(c) \neq \mathbf{L}
\end{array}
$$

with k a context, and ρ a renaming map.

Branch Removal - s_3 I

$$\frac{\Gamma(e) \nsubseteq L_{@p} \quad \begin{array}{r@{\;}l} x_1 &= \mathtt{fresh}(x) \\ x_2 &= \mathtt{fresh}(x) \\ \rho_1' &= \rho_1[x \mapsto x_1] \\ \rho_2' &= \rho_2[x \mapsto x_2] \end{array}}{\rho_1, \rho_2, x = e \rightharpoonup_{@p} \rho_1', \rho_2', (x_1 = \rho_1(e); x_2 = \rho_2(e))}$$

$$\frac{\Gamma(e_1) \cup \Gamma(e_2) \nsubseteq L_{@p}}{\rho_1, \rho_2, t[e_1] = e_2 \rightharpoonup_{@p} \rho_1, \rho_2, \left(\begin{array}{c} t[\rho_1(e_1)] = k?\rho_1(e_2) : t[\rho_1(e_1)]; \\ t[\rho_2(e_1)] = k?\rho_2(e_2) : t[\rho_2(e_1)] \end{array} \right)}$$

$$\frac{\begin{array}{c} \rho_1, \rho_2, s_1 \rightharpoonup_{@p} \rho_1', \rho_2', s_1' \\ \rho_1', \rho_2', s_2 \rightharpoonup_{@p} \rho_1'', \rho_2'', s_2' \end{array}}{\rho_1, \rho_2, (s_1; s_2) \rightharpoonup_{@p} \rho_1'', \rho_2'', (s_1'; s_2')}$$

with $L_{@p} = L \cup \{I(p'), p' \neq p\}$

Branch Removal - s_3 II

$$\frac{\begin{array}{c}
\Gamma(c) \subseteq L_{@p} \\
\begin{array}{cc}
\rho_1, \rho_2, s_1 \rightarrowtriangle_{@p} \rho'_1, \rho'_2, s'_1 & \rho_1, \rho_2, s_2 \rightarrowtriangle_{@p} \rho''_1, \rho''_2, s'_2 \\
\dot{\rho}_1 \gtrsim \rho'_1 & \dot{\rho}_1 \gtrsim \rho''_1 \\
\dot{\rho}_2 \gtrsim \rho'_2 & \dot{\rho}_2 \gtrsim \rho''_2
\end{array} \\
\begin{array}{rcl}
s_{1i} & = & s'_1 + (\dot{\rho}_1(x) = \rho'_1(x) | x \in \dot{\rho}_1) \\
s_{1f} & = & s_{1i} + (\dot{\rho}_2(x) = \rho'_2(x) | x \in \dot{\rho}_2) \\
s_{2i} & = & s'_2 + (\dot{\rho}_1(x) = \rho''_1(x) | x \in \dot{\rho}_1) \\
s_{2f} & = & s_{2i} + (\dot{\rho}_2(x) = \rho''_2(x) | x \in \dot{\rho}_2)
\end{array} \\
\dot{\rho}_1, \dot{\rho}_2, s_3 \rightarrowtriangle_{@p} \ddot{\rho}_1, \ddot{\rho}_2, s'_3
\end{array}}{\rho_1, \rho_2, \textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2 \ \textbf{ next } s_3 \ \rightarrowtriangle_{@p} \ddot{\rho}_1, \ddot{\rho}_2, \textbf{if } c \textbf{ then } s_{1f} \textbf{ else } s_{2f} \ \textbf{ next } s'_3}$$

where $\rho_1 \gtrsim \rho_2 \iff \texttt{keys}(\rho_2) \subset \texttt{keys}(\rho_1)$

Branch Removal - s_3 III

$$\frac{\begin{array}{c} \rho'_1, \rho'_2, s \twoheadrightarrow_{@p} \rho''_1, \rho''_2, s' \\ \rho'_1 > \rho_1 \quad \rho''_1 \simeq \rho'_1 \\ \rho'_2 > \rho_2 \quad \rho''_2 \simeq \rho'_2 \\ \rho'_1 \cap \rho'_2 = \emptyset \\ s' += (\rho'_1(x) = \rho''_1(x) | x \in \rho''_1 \Delta \rho'_1) \\ s' += (\rho'_2(x) = \rho''_2(x) | x \in \rho''_2 \Delta \rho'_2) \end{array}}{\rho_1, \rho_2, \texttt{for}\ i\ \texttt{in}\ (e_1, e_2)\ \texttt{do}\ s \twoheadrightarrow_{@p} \rho'_1, \rho'_2, s'}$$

where Δ is the symmetric difference operator, and
$\rho_1 \simeq \rho_2 \iff \texttt{keys}(\rho_1) = \texttt{keys}(\rho_2)$

Limitations

- As our language doesn't allow early returns, we do not have a return deferral policy
- Function calls are not handled, and supposed to be inlined by previous passes of the compiler
- We suppose operators do not leak timing information. Refining the leaky semantics and adapting the type system would allow it.
- While loops, both High and Low are not handled by now, because of non-termination problem.

Conclusion

- Handles a broader class of program than FaCT
- Being implemented as a pass in the JASMIN compiler

What about the future ?

Conclusion

- Handles a broader class of program than FaCT
- Being implemented as a pass in the JASMIN compiler

What about the future ?

- Handle functions calls
- Prove both semantic conservation, and constant-time characteristic
- Test efficiency on real-world programs