Utrecht University

Constant Time Cryptography

Ivo Gabe de Wolff and Marco Vassena

Deadline: Wednesday, 24 May, 23:59

The goal of this assignment is to implement an analysis that enforces the constant-time discipline for a functional language. As discussed in the lectures, the analysis rejects programs whose execution time or memory access pattern depend on secret information. The analysis operates on a functional language with mutable arrays and strict evaluation (call-by-value) given by the following syntax:

$e ::= n$	*Natural numbers*
$\mid \textbf{true} \mid \textbf{false}$	*Booleans*
$\mid x$	*Variables*
$\mid \textbf{fn}\ x \to e$	*Function*
$\mid \textbf{fun}\ f\ x \to e$	*Recursive function*
$\mid \textbf{let}\ x\ =\ e\ \textbf{in}\ e$	*Let*
$\mid e\ e$	*Application*
$\mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$	*If-then-else*
$\mid e \oplus e$	*Binary operators*
$\mid e :: \hat{\tau}$	*Type annotation*
$\mid e; e$	*Sequence*
$\mid \textbf{array}\ e\ e$	*Array allocation*
$\mid e[e]$	*Reading from an array*
$\mid e[e] = e$	*Writing to an array*
$\mid (e, e)$	*Pair*
$\mid \textbf{case}\ e\ \textbf{of}\ (x, y) \to e$	*Pattern match on pair*
$\mid []$	*Empty list*
$\mid e : e$	*Cons*
$\mid \textbf{case}\ e\ \textbf{of}\ [] \to e, x : y \to e$	*Pattern match on list*

Arrays are constructed using $\textbf{array}\ e_1\ e_2$, where e_1 is an expression computing the (public) size of the array, and e_2 the default value, with which the array is initialised. Array accesses are bounds checked. An array write, $e_1[e_2] = e_3$, returns the right hand side (e_3) after performing the update.

The language has various binary operators $\oplus$. All operators but division ($/$) are constant time. The logical operators are strict; the second argument is always evaluated. Operators $==$ and $!=$ can be used on values of any type, as long as the two arguments have the same type.

$$
\begin{array}{rl}
\oplus ::= + \mid - \mid * \mid / & \textit{Arithmetic operators} \\
\mid \&\& \mid \mid\mid & \textit{Logical operators} \\
\mid == \mid != \mid < & \textit{Comparison operators}
\end{array}
$$

The annotated type system has the following types:

$$
\begin{array}{rl}
\ell ::= L \mid H & \textit{Labels} \\
\hat{\tau} ::= \mathrm{Nat} & \textit{Natural number} \\
\mid \mathrm{Bool} & \textit{Boolean} \\
\mid \hat{\tau}^{\ell} \mapsto \hat{\tau}^{\ell} & \textit{Function} \\
\mid \mathrm{Array}\ \hat{\tau}^{\ell} & \textit{Array} \\
\mid (\hat{\tau}^{\ell}, \hat{\tau}^{\ell}) & \textit{Pair} \\
\mid \mathrm{List}\ \hat{\tau}^{\ell} & \textit{List} \\
\mid \alpha & \textit{Type variable}
\end{array}
$$

Values are marked either private (high confidential, H) or public (low confidential, L). Type judgements are of the form $\hat{\Gamma} \vdash_{\mathsf{CTC}} e : \hat{\tau}^{\ell}$, saying that expression e has type $\hat{\tau}$ and label ℓ.

Use the techniques explained in the lectures on Control Flow Analysis to track of the confidentiality level of values, including function arguments and results. The template for this assignment is available via GitHub Classrooms. The template already provides the AST and parser for this language. It defines the data types for the type system: **data** TypeScheme corresponds to σ, **data** LabelledType corresponds to $\hat{\tau}^{\ell}$ and **data** Type to $\hat{\tau}$. You can implement the analysis in src/Analysis.hs.

Steps

1. *(0.8pt)* Implement type inference for the underlying type system (without polymorphism, arrays, pairs and lists). Try to follow the course material (slides) as closely as possible.
2. *(0.7pt)* Extend the analysis with polymorphism for the underlying type system.
3. *(0.5pt)* Extend the analysis with confidentiality labels ℓ to the type system.
4. *(0.5pt)* Report errors for violations of constant time.
5. *(0.6pt)* Implement subeffecting (subtyping on the outer level of an annotated type). Provide 3 small example programs that benefit from subeffecting: they should not compile without subeffecting, and pass with subeffecting.
6. *(1.3pt)* Add mutable arrays to your analysis. Note that the size of an array must always be public. Provide 3 small example programs for arrays, of which at least one compiles and one violates constant time.
7. *(0.8pt)* Add pairs to your analysis. Provide 3 small example programs for pairs, of which at least one passes and one failes.
8. *(1.3pt)* Add lists to your analysis. Provide 3 small example programs for lists, of which at least one passes and one failes.
9. *(1.0pt)* In the report, provide all typing rules related to subeffecting, function application, pairs and lists.
10. *(1.0pt)* Code readability is worth one point.

Bonus

You may implement any of the following additions for further points, preferably after completing all previous step. These may require some more work or research, so it may be useful to discuss them with the teachers during the labs. You can choose which bonus you like most. By implementing two additions you can already score a 10.

1. *(1pt)* Extend subeffecting to subtyping. The report should formalise this with typing rules. You should provide a set of example programs which benefit from this. Make sure you handle co-, contra- and invariance correctly.
2. Type annotations:
   (a) *(1.0pt)* Allow type annotation without labels. Whereas the template only supports types with labels, for this bonus you will need to allow that some types do not have labels. These should be inferred by the analysis. This bonus might require more changes to the parser than you think.
   (b) *(1.0pt)* Implement type annotations on let-bindings. This is worth 1.0pt if you support type schemes, or 0.3pt if you only support types without quantifications.
   (c) *(1.0pt)* Implement type annotations on recursive functions and with support for *polymorphic recursion*. In $\textbf{fun}\ f\ x \to e$, variable f may now be annotated with a *type scheme* and it should be possible to instantiate that different in each recursive call.
3. Implement a program transformation:
   (a) *(1pt)* A transformation which annotates all bindings of variables (in let expressions and functions) with their type.
   (b) *(1pt)* A transformation which converts the standard if-then-else ($\textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2$) to a strict if-then-else, written as $\textbf{let}\ y = e_1 \textbf{in let}\ z = e_2\ \textbf{in if!}\ x\ \textbf{then}\ y\ \textbf{else}\ z$. The strict if-then-else always evaluates both alternatives, and returns one of them without branching (via *CMOV* in x86 or *select* in LLVM), thus guaranteeing constant time. This transformation should happen if (1) both alternatives have no side effects and no function calls and (2) the condition is secret. Instead of giving an error, your program will perform this transformation. If condition (1) doesn't hold, you should of course still give an error if (2) holds.
   (c) *(1pt)* In combination with one of the previous two bonusses: implement the data type for the IR before and after the transformation with the techniques from Trees That Grow[1].
4. *(1pt)* Implement polyvariance: polymorphism over annotation variables. The report should formalise this with typing rules.

Submission

The deliverables of this project are (1) the code of the implementation, (2) a set of examples that demonstrate that your analysis works and (3) a short report including the grammar for annotated types, typing rules and the specification of what functionality is implemented. Include your student numbers in the report.

Submission of the practical happens via GitHub Classrooms: `https://classroom.github.com/a/5yJO_Yr3`

[1] `https://arxiv.org/abs/1610.04799`