Output-sensitive Information flow analysis

Cristian Ene, Laurent Mounier, Marie-Laure Potet

Output-sensitive Information flow analysis*

Cristian Ene[0000−0001−6322−0383], Laurent Mounier[0000−0001−9925−098X], and Marie-Laure Potet[0000−0002−7070−6290]

Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France
Firstname.Lastname@univ-grenoble-alpes.fr

Abstract. *Constant-time* programming is a countermeasure to prevent cache based attacks where programs should not perform memory accesses that depend on secrets. In some cases this policy can be safely relaxed if one can prove that the program does not leak more information than the public outputs of the computation.
We propose a novel approach for verifying constant-time programming based on a new information flow property, called *output-sensitive noninterference*. Noninterference states that a public observer cannot learn anything about the private data. Since real systems need to intentionally declassify some information, this property is too strong in practice. In order to take into account public outputs we proceed as follows: instead of using complex explicit declassification policies, we partition variables in three sets: input, output and leakage variables. Then, we propose a typing system to statically check that leakage variables do not leak *more information about the secret inputs than the public normal output*. The novelty of our approach is that we track the dependence of leakage variables with respect not only to the initial values of input variables (as in classical approaches for noninterference), but taking also into account the final values of output variables. We adapted this approach to LLVM IR and we developed a prototype to verify LLVM implementations.

Keywords: Information flow · Output-sensitive non-interference · Type system.

1 Introduction

An important task of cryptographic research is to verify cryptographic implementations for security flaws, in particular to avoid so-called timing attacks. Such attacks consist in measuring the execution time of an implementation on its execution platform. For instance, Brumley and Boneh [12] showed that it was possible to mount remote timing attacks by against OpenSSL's implementation of the RSA decryption operation and to recover the key. Albrecht and Paterson [3] showed that the two levels of protection offered against the Lucky 13 attack from [2] in the first release of the new implementation of TLS were imperfect. A

* This work is supported by the French National Research Agency in the framework of the "Investissements d' avenir" program (ANR-15-IDEX-02)

related class of attacks are *cache-based attacks* in which a malicious party is able to obtain memory-access addressses of the target program which may depend on secret data through observing cache accesses. Such attacks allow to recover the complete AES keys [17].

A possible countermeasure is to follow a very strict programming discipline called **constant-time programming**. Its principle is to avoid branchings controlled by secret data and memory load/store operations indexed by secret data. Recent secure C libraries such as NaCl [10] or mbedTLS[1] follow this programming discipline. Until recently, there was no rigorous proof that constant-time algorithms are protected to cache-based attacks. Moreover, many cryptographic implementations such as PolarSSL AES, DES, and RC4 make array accesses that depend on secret keys and are not constant time. Recent works [6, 4, 11] fill this gap and develop the first formal analyzes that allow to verify if programs are correct with respect to the constant-time paradigm.

An interesting extension was brought by Almeida et al. [4] who enriched the constant-time paradigm *"distinguishing not only between public and private input values, but also between private and publicly observable output values"*. This distinction raises interesting technical and theoretical challenges. Indeed, constant-time implementations in cryptographic libraries like OpenSSL include optimizations for which paths and addresses can depend not only on public input values, but also on publicly observable output values. Hence, considering only input values as non-secret information would thus incorrectly characterize those implementations as non-constant-time. [4] also develops a verification technique based on *symbolic execution*. Howvere, the soundness of their approach depends in practice on the soundness of the underlying symbolic execution engine, which is very difficult to guarantee for real-world programs with loops. Moreover, their product construction can be very expensive in the worst case.

In this paper we deal with *statically checking programs* for **output-sensitive constant-time** correctness: programs can still do branchings or memory accesses controlled by secret data if the information that is leaked is subsumed by the normal output of the program. To give more intuition about the property that we want to deal with, let us consider the following example, where *ct_eq* is a constant time function that allows to compare the arguments:

```
good = 1;
for (i=0; i<B_Size; i++){good = good & ct_eq(secret[i],in_p[i]);}
if (!good) { for(i=0; i<B_Size; i++) secret[i] = 0; }
return good;
```

Let suppose that the array variable *secret* is secret, and all the other variables are public. Intuitively this a sort of one-time check password verifying that $in\_p = secret$ and otherwise overwrites the array *secret* with zero. Obviously, this function is not constant-time as the variable *good* depends on *secret*, and hence branching on *good* violates the principles of constant-time programming.

[1] mbed TLS (formerly known as PolarSSL). https://tls.mbed.org/

It is easy to transform this program into an equivalent one which is constant time. For example one could replace

```
if (!good) { for(i=0; i<B_Size; i++) secret[i] = 0; }
```

by

```
for (i=0; i<B_Size; i++) {secret[i] = secret[i] & ct_eq(good,1);}
```

But branching on *good* is a benign optimization, since anyway, the value of *good* is the normal output of the program. Hence, even if the function is not constant-time, it should be considered **output-sensitive constant time** with respect to its specification. Such optimization opportunities arise whenever the interface of the target application specifies what are the publicly observable outputs, and this information is sufficient to classify the extra leakage as benign [4].

The objective of this work is to propose a *static method* to check if a program is *output-sensitive constant time secure*. We emphasize that our goal is **not** to verify that the legal output leaks "too much", but rather to ensure that the unintended (side-channel) output does not leak **more than** this legal output.

First, we propose a novel approach for verifying constant-time security based on a new information flow property, called *output-sensitive noninterference*. Information-flow security prevents confidential information to be leaked to public channels. Noninterference states that a public observer cannot learn anything about the private data. Since real systems need to intentionally declassify some information, this property is too strong. An alternative is *relaxed noninterference* which allows to specify explicit *downgrading policies*. In order to take into account public outputs while staying independent of how programs intentionally declassify information, we develop an alternative solution: instead of using complex explicit policies for functions, we partition variables in three sets: input, output and *leakage variables*. Hence we distinguish between the legal public output and the information that can leak through side-channels, expressed by adding fresh additional leakage variables. Then we propose a typing system that can statically check that leakage variables do not leak more secret information than the public normal output. The novelty of our approach is that we track the dependence of leakage variables with respect to both the *initial value of input variables* (as classically the case for noninterference) and *final values of output variables*. Then, we show how to verify that a program written in a high-level language is output-sensitive constant time secure by using this typing system.

Since timed and cache-based attacks target the executions of programs, it is important to carry out this verification in a language close to the machine-executed assembly code. Hence, we adapt our approach to a generic unstructured assembly language inspired from LLVM and we show how we can verify programs coded in LLVM. Finally, we developed a prototype tool implementing our type system and we show how it can be used to verify LLVM implementations.

To summarize, this work makes the following contributions described above: - in section 2 we reformulate output-sensitive constant-time as a new interesting noninterference property and we provide a sound type system that guarantees

$$\frac{}{(x := e, \sigma) \longrightarrow \sigma[x \mapsto \sigma(e)]} \qquad \frac{}{(\mathsf{skip}, \sigma) \longrightarrow \sigma}$$

$$\frac{(c_1, \sigma) \longrightarrow \sigma'}{(c_1; c_2, \sigma) \longrightarrow (c_2, \sigma')} \qquad \frac{(c_1, \sigma) \longrightarrow (c_1', \sigma')}{(c_1; c_2, \sigma) \longrightarrow (c_1'; c_2, \sigma')}$$

$$\frac{\sigma(e) = 1 \ ?\ i = 1 :\ i = 2}{(\mathsf{If}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ , \sigma) \longrightarrow (c_i, \sigma)} \qquad \frac{\sigma(e) \neq 1}{(\mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}\ , \sigma) \longrightarrow \sigma}$$

$$\frac{\sigma(e) = 1}{(\mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}\ , \sigma) \longrightarrow (c; \mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}\ , \sigma)}$$

Fig. 1. Operational semantics of the While language

that well-typed programs are output-sensitive noninterferent;
- in section 3 we show that this general approach can be used to verify that programs written in a high-level language are output-sensitive constant time;
- in section 4 we adapt our approach to the LLVM-IR language and we develop a prototype tool that can be used to verify LLVM implementations.

An extended version of this paper, inluding all proofs and complete type systems is available on-line[2].

2 Output-sensitive non-interference

2.1 The While language and Output-sensitive noninterference

In order to reason about the security of the code, we first develop our framework in *While*, a simple high-level structured programming language. In section 3 we shall enrich this simple language with arrays and in section 4 we adapt our approach to a generic unstructured assembly language. The syntax of While programs is listed below:

$$c ::= x := e \mid \mathsf{skip} \mid c_1; c_2 \mid \mathsf{If}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \mid \mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}$$

Meta-variables x, e and c range over the sets of program variables Var, expressions and programs, respectively. We leave the syntax of expressions unspecified, but we assume they are deterministic and side-effect free. The semantics is shown in Figure 1. The reflexive and transitive closure of $\longrightarrow$ is denoted by $\Longrightarrow$. A state σ maps variables to values, and we write $\sigma(e)$ to denote the value of expression e in state σ. A configuration (c, σ) is a program c to be executed along with the current state σ. Intuitively, if we want to model the security of some program c with respect to side-channel attacks, we can assume that there are three special

[2] https://www-verimag.imag.fr/∼Cristian.Ene/OSNI/main.pdf

subsets of variables: X_I the public input variables, X_O the public output variables and X_L the variables that leak information to some malicious adversary. Then, output sensitive nonintereference asks that every two complete executions starting with X_I-equivalent states and ending with X_O-equivalent final states must be indistinguishable with respect to the leakage variables X_L .

Definition 1. *(adapted from [4]) Let $X_I, X_O, X_L \subseteq Var$ be three sets of variables, intended to represent the input, the output and the leakage of a program. A program c is* **(X_I, X_O, X_L)-secure** *when all its executions starting with X_I-equivalent stores and leading to X_O-equivalent final stores, give X_L-equivalent final stores. Formally, for all $\sigma, \sigma', \rho, \rho'$, if $\langle c, \sigma\rangle \Longrightarrow \sigma'$ and $\langle c, \rho\rangle \Longrightarrow \rho'$ and $\sigma =_{X_I} \rho$ and $\sigma' =_{X_O} \rho'$, then $\sigma' =_{X_L} \rho'$.*

2.2 Typing rules

This section introduces a type-based information flow analysis that allows to check whether a While program is output-sensitive noninterferent, i.e. the program does not leak more information about the secret inputs than the public normal output.

As usual, we consider a flow lattice of security levels $\mathcal{L}$. An element x of $\mathcal{L}$ is an atom if $x \neq \bot$ and there exists no element $y \in \mathcal{L}$ such that $\bot \sqsubset y \sqsubset x$. A lattice is called *atomistic* if every element is the join of atoms below it.

Assumption 2.21 *Let $(\mathcal{L}, \sqcap, \sqcup, \bot, \top)$ be an atomistic continuous bounded lattice. As usual, we denote $t_1 \sqsubseteq t_2$ iff $t_2 = t_1 \sqcup t_2$. We assume that there exists a distinguished subset $\mathcal{T}_O \subseteq \mathcal{L}$ of atoms.*

Hence, from the above assumption, for any $\tau_o \in \mathcal{T}_O$ and for any $t_1, t_2 \in \mathcal{L}$:
1. $\tau_o \sqsubseteq t_1 \sqcup t_2$ implies $\tau_o \sqsubseteq t_1$ or $\tau_o \sqsubseteq t_2$,
2. $\tau_o \sqsubseteq t_1$ implies that there exists $t \in \mathcal{L}$ such that $t_1 = t \sqcup \tau_o$ and $\tau_0 \not\sqsubseteq t$.

A type environment $\Gamma : Var \mapsto \mathcal{L}$ describes the security levels of variables and the dependency with respect to the current values of variables in X_O. In order to catch dependencies with respect to current values of output variables, we associate to each output variable $o \in X_O$ a fixed and unique symbolic type $\alpha(o) \in \mathcal{T}_O$. For example if some variable $x \in Var$ has the type $\Gamma(x) = Low \sqcup \alpha(o)$, it means that the value of x depends only on public input and the current value of the output variable $o \in X_O$.

Hence, we assume that there is a fixed injective mapping $\alpha : X_0 \mapsto \mathcal{T}_0$ such that $\bigwedge_{o_1, o_2 \in X_O} \big(o_1 \neq o_2 \Rightarrow \alpha(o_1) \neq \alpha(o_2)\big) \wedge \bigwedge_{o \in X_O} \big(\alpha(o) \in \mathcal{T}_O\big)$. We extend mappings Γ and α to sets of variables in the usual way: given $A \subseteq Var$ and $B \subseteq X_O$ we note $\Gamma(A) \stackrel{def}{=} \bigsqcup_{x \in A} \Gamma(x)$, $\alpha(B) \stackrel{def}{=} \bigsqcup_{x \in B} \alpha(x)$.

Our type system aims to satisfy the following output sensitive non-interference condition: if the *final* values of output variables in X_O remain the same, only changes to *initial* inputs with types $\sqsubseteq t$ should be visible to *leakage* outputs with type $\sqsubseteq t \sqcup \alpha(X_O)$. More precisely, given a derivation $\vdash_\alpha \Gamma\{c\}\Gamma'$, the final value

of a variable x with final type $\Gamma'(x) = t \sqcup \alpha(A)$ for some $t \in \mathcal{L}$ and $A \subseteq X_O$, should depend at most on the initial values of those variables y with initial types $\Gamma(y) \sqsubseteq t$ and on the final values of variables in A. We call "real dependencies" the dependencies with respect to initial values of variables and "symbolic dependencies" the dependencies with respect to the current values of output variables. Following [19] we formalize the non-interference condition satisfied by the typing system using reflexive and symmetric relations.

We write $=_{A_0}$ for relation which relates mappings which are equal on all values in A_0 i.e. for two mappings $f_1, f_2 : A \mapsto B$ and $A_0 \subseteq A$, $f_1 =_{A_0} f_2$ iff $\forall a \in A_0, f_1(a) = f_2(a)$. For any mappings $f_1 : A_1 \mapsto B$ and $f_2 : A_2 \mapsto B$, we write $f_1[f_2]$ the operation which updates f_1 according to f_2, namely $(f_1[f_2])(x) =$ if $x \in A_2$ then $f_2(x)$ else $f_1(x)$. Given $\Gamma : Var \mapsto \mathcal{L}$, $X \subseteq Var$ and $t \in \mathcal{L}$, we write $=_{\Gamma,X,t}$ for the reflexive and symmetric relation which relates states that are equal on all variables having type $v \sqsubseteq t$ in environment Γ, provided that they are equal on all variables in X: $\sigma =_{\Gamma,X,t} \sigma'$ iff $\sigma =_X \sigma' \Rightarrow \big(\forall x, (\Gamma(x) \sqsubseteq t \Rightarrow \sigma(x) = \sigma'(x))\big)$. When $X = \emptyset$, we omit it, hence we write $=_{\Gamma,t}$ instead of $=_{\Gamma,\emptyset,t}$.

Definition 2. *[20] Let $\mathcal{R}$ and $\mathcal{S}$ be reflexive and symmetric relations on states. We say that program c maps $\mathcal{R}$ into $\mathcal{S}$, written $c : \mathcal{R} \Longrightarrow \mathcal{S}$, iff $\forall \sigma, \rho$, if $\langle c, \sigma\rangle \Longrightarrow \sigma'$ and $\langle c, \rho\rangle \Longrightarrow \rho'$ then $\sigma \mathcal{R} \rho \Rightarrow \sigma' \mathcal{S} \rho'$.*

The type system we propose enjoys the following useful property:

if $\vdash_\alpha \Gamma\{c\}\Gamma'$ then c: $=_{\Gamma,\Gamma(X_I)} \implies =_{\Gamma',X_O,\alpha(X_O)\sqcup\Gamma(X_I)}$

This property is an immediate consequence of Theorem 2.

Hence, in order to prove that the above program c is output sensitive non-interferent according to Definition 1, it is enough to check that for all $x_l \in X_L$, $\Gamma'(x_l) \sqsubseteq \alpha(X_O) \sqcup \Gamma(X_I)$. Two executions of the program c starting from initial states that coincide on input variables X_I, and ending in final states that coincide on output variables X_O, will coincide also on the leaking variables X_L.

We now formally introduce our typing system. Due to assignments, values and types of variables change dynamically. For example let us assume that at some point during the execution, the value of x depends on the initial value of some variable y and the current value of some output variable o (which itself depends on the initial value of some variable h), formally captured by an environment Γ where $\Gamma(o) = \Gamma_0(h)$ and $\Gamma(x) = \Gamma_0(y) \sqcup \alpha(o)$, where Γ_0 represents the initial environment. If the next to be executed instruction is some assignment to o, then the current value of o will change, so we have to mirror this in the new type of x: even if the value of x does not change, its new type will be $\Gamma'(x) = \Gamma_0(y) \sqcup \Gamma_0(h)$ (assuming that $\alpha(o) \not\sqsubseteq \Gamma_0(y)$). Hence $\Gamma'(x)$ is obtained by replacing in $\Gamma(x)$ the symbolic dependency $\alpha(o)$ with the real dependency $\Gamma(o)$.

Definition 3. *If $t^0 \in \mathcal{T}_O$ is an atom and $t', t \in \mathcal{L}$ are arbitrary types, then we denote by $t[t'/t^0]$ the type obtained by replacing (if any) the occurrence of t^0 by t' in the decomposition in atoms of t. Now we extend this definition to environments: let $x \in X_O$ and $p \in \mathcal{L}$. Then $\Gamma_1 \stackrel{def}{=} \Gamma \lhd_\alpha x$ represents the environment where the symbolic dependency on the last value of x of all variables is replaced by the real type of x: $\Gamma_1(y) \stackrel{def}{=} (\Gamma(y))[\Gamma(x)/\alpha(x)]$. Similarly, $(p, \Gamma) \lhd_\alpha x \stackrel{def}{=} p[\Gamma(x)/\alpha(x)]$.*

$$\text{As1}\ \frac{x \notin X_O}{p \vdash_\alpha \Gamma\{x := e\}\Gamma[x \mapsto p \sqcup \Gamma[\alpha](fv(e))]} \qquad \text{As2}\ \frac{x \in X_O \setminus fv(e) \qquad \Gamma_1 = \Gamma \lhd_\alpha x}{p \vdash_\alpha \Gamma\{x := e\}\Gamma_1[x \mapsto p \sqcup \Gamma_1[\alpha](fv(e))]}$$

$$\text{Skip}\ \frac{}{p \vdash_\alpha \Gamma\{skip\}\Gamma} \qquad \text{As3}\ \frac{x \in X_O \cap fv(e) \qquad \Gamma_1 = \Gamma \lhd_\alpha x}{p \vdash_\alpha \Gamma\{x := e\}\Gamma_1[x \mapsto p \sqcup \Gamma(x) \sqcup \Gamma_1[\alpha](fv(e) \setminus x)]}$$

$$\text{Seq}\ \frac{p \vdash_\alpha \Gamma\{c_1\}\Gamma_1 \qquad p \vdash_\alpha \Gamma_1\{c_2\}\Gamma_2}{p \vdash_\alpha \Gamma\{c_1; c_2\}\Gamma_2} \qquad \text{Sub}\ \frac{p_0 \sqsubseteq p_1 \quad \Gamma \sqsubseteq \Gamma' \qquad p_1 \vdash_\alpha \Gamma'\{c\}\Gamma_1' \quad \Gamma_1' \sqsubseteq \Gamma_1}{p_0 \vdash_\alpha \Gamma\{c\}\Gamma_1}$$

$$\text{If}\ \frac{\begin{array}{c} p' = (\Gamma[\alpha](fv(e)), \Gamma) \lhd_\alpha (\mathbf{aff^O}(c_1) \cup \mathbf{aff^O}(c_2)) \\ p \sqcup p' \vdash_\alpha \Gamma\{c_i\}\Gamma_i \qquad \Gamma' = \Gamma_1 \lhd_\alpha \mathbf{aff^O}(c_2) \sqcup \Gamma_2 \lhd_\alpha \mathbf{aff^O}(c_1) \end{array}}{p \vdash_\alpha \Gamma\{\mathsf{If}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ \}\Gamma'}$$

$$\text{Wh}\ \frac{\begin{array}{c} p_e = (\Gamma[\alpha](fv(e)), \Gamma) \lhd_\alpha \mathbf{aff^O}(c) \\ p \sqcup p_e \vdash_\alpha \Gamma\{c\}\Gamma' \qquad \Gamma' \sqcup (\Gamma \lhd_\alpha \mathbf{aff^O}(c)) \sqsubseteq \Gamma \end{array}}{p \vdash_\alpha \Gamma\{\mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}\ \}\Gamma}$$

Fig. 2. Flow-sensitive typing rules for commands with output

We want now to extend the above definition from a single output variable x to subsets $X \subseteq X_O$. Our typing system will ensure that each generated environment Γ will not contain circular symbolic dependencies between output variables, i.e., there are no output variable $o_1, o_2 \in X_O$ such that $\alpha(o_1) \sqsubseteq \Gamma(o_2)$ and $\alpha(o_2) \sqsubseteq \Gamma(o_1)$. We can associate a graph $\mathcal{G}(\Gamma) = (X_O, E)$ to an environment Γ, such that $(o_1, o_2) \in E$ iff $\alpha(o_1) \sqsubseteq \Gamma(o_2)$. We say that Γ is **well formed**, denoted $\mathcal{AC}(\Gamma)$, if $\mathcal{G}(\Gamma)$ is an acyclic graph. For acyclic graphs $\mathcal{G}(\Gamma)$ we extend Definition 3 to subsets $X \subseteq X_O$, by first fixing an ordering $X = \{x_1, x_2, \ldots x_n\}$ of variables in V compatible with the graph (i.e. $j \leq k$ implies that there is no path from x_k to x_j), and then $(p, \Gamma) \lhd_\alpha X \stackrel{def}{=} (((p, \Gamma) \lhd_\alpha x_1) \lhd_\alpha x_2) \ldots \lhd_\alpha x_n$.

Let $\mathbf{aff}(c)$ be the set of assigned variables in a program c and let us denote $\mathbf{aff^I}(c) \stackrel{def}{=} \mathbf{aff}(c) \cap (Var \setminus X_O)$ and $\mathbf{aff^O}(c) \stackrel{def}{=} \mathbf{aff}(c) \cap X_O$. We define the ordering over environments: $\Gamma_1 \sqsubseteq \Gamma_2 \stackrel{def}{=} \bigwedge_{x \in Var} \Gamma_1(x) \sqsubseteq \Gamma_2(x)$. For a command c, judgements have the form $p \vdash_\alpha \Gamma\{c\}\Gamma'$ where $p \in \mathcal{L}$ and Γ and Γ' are type environments well-formed. The inference rules are shown in Figure 2. The idea is that if Γ describes the security levels of variables which hold before execution of c, then Γ' will describe the security levels of those variables after execution of c. The type p represents the usual program counter level and serves to eliminate indirect information flows; the derivation rules ensure that all variables that can be changed by c will end up (in Γ') with types greater than or equal to p. As usual, whenever $p = \bot$ we drop it and write $\vdash_\alpha \Gamma\{c\}\Gamma'$ instead of $\bot \vdash_\alpha \Gamma\{c\}\Gamma'$. Throughout this paper the type of an expression e is defined simply by taking the lub of the types of its free variables $\Gamma[\alpha](fv(e))$, for example the type of $x+y+o$

$$p = \bot, \quad \Gamma_0 = [y \to Y, z \to Z, o_1 \to O_1, o_2 \to O_2]$$

(1) $o_1 := x + 1$

$$\Gamma_1 = [y \to Y, z \to Z, \mathbf{o_1} \to \mathbf{X}, o_2 \to O_2]$$

(2) $y := o_1 + z$

$$\Gamma_2 = [\mathbf{y} \to \mathbf{\overline{O_1} \sqcup Z}, z \to Z, o_1 \to X, o_2 \to O_2]$$

(3) $o_1 := u$

$$\Gamma_3 = [\mathbf{y} \to \mathbf{X \sqcup Z}, z \to Z, \mathbf{o_1} \to \mathbf{U}, o_2 \to O_2]$$

(4) $z := o_1 + o_3$

$$\Gamma_4 = [y \to X \sqcup Z, \mathbf{z} \to \mathbf{\overline{O_1} \sqcup \overline{O_3}}, o_1 \to U, o_2 \to O_2]$$

(5) If $(o_2 = o_3 + x)$ $\quad \mathbf{p = \overline{O_3} \sqcup O_2 \sqcup X}$

(6) then $o_1 := o_2$

$$\Gamma_6 = [y \to X \sqcup Z, \mathbf{z} \to \mathbf{U \sqcup \overline{O_3}}, \mathbf{o_1} \to \mathbf{\overline{O_3} \sqcup O_2 \sqcup X \sqcup \overline{O_2}}, o_2 \to O_2]$$

(7) else $o_2 := o_1$

$$\Gamma_7 = [y \to X \sqcup Z, z \to \overline{O_1} \sqcup \overline{O_3}, o_1 \to U, o_2 \to \mathbf{\overline{O_3} \sqcup O_2 \sqcup X \sqcup \overline{O_1}}]$$

(8) fi

$$\Gamma_8 = (\Gamma_6 \lhd_\alpha o_2) \sqcup (\Gamma_7 \lhd_\alpha o_1) = [y \to X \sqcup Z, \mathbf{z} \to \mathbf{U \sqcup \overline{O_3}},$$
$$o_1 \to \mathbf{\overline{O_3} \sqcup O_2 \sqcup X \sqcup U}, o_2 \to \mathbf{\overline{O_3} \sqcup O_2 \sqcup X \sqcup U}]$$

Fig. 3. Example of application for our typing system

where o is the only output variable is $\Gamma(x) \sqcup \Gamma(y) \sqcup \alpha(o)$. This is consistent with the typings used in many systems, though more sophisticated typing rules for expressions would be possible in principle. Notice that considering the type of an expression to be $\Gamma[\alpha](fv(e))$ instead of $\Gamma(fv(e))$ allows to capture the dependencies with respect to the current values of output variables. In order to give some intuition about the rules, we present a simple example in Figure 3.

Example 1. Let $\{x, y, z, u\} \subseteq Var \setminus X_O$ and $\{o_1, o_2, o_3\} \subseteq X_O$ be some variables, and let us assume that $\forall i \in \{1, 2, 3\}$, $\alpha(o_i) = \overline{O_i}$. We assume that the initial environment is $\Gamma_0 = [x \to X, y \to Y, z \to Z, u \to U, o_1 \to O_1, o_2 \to O_2, o_3 \to O_3]$. Since the types of variables x, u and o_3 do not change, we omit them in the following. We highlighted the changes with respect to the previous environment. After the first assignment, the type of o_1 becomes X, meaning that the current value of o_1 depends on the initial value of x. After the assignment $y := o_1 + z$, the type of y becomes $\overline{O_1} \sqcup Z$, meaning that the current value of y depends on the initial value of z and the current value of o_1. After the assignment $o_1 = u$, the type of y becomes $X \sqcup Z$ as o_1 changed and we have to mirror this in the dependencies of y, and the type of o_1 becomes X. When we enter in the If , the program counter level changes to $p = \overline{O_3} \sqcup O_2 \sqcup X$ as the expression $o_2 = o_3 + x$ depends on the values of variables o_2, o_3, x, but o_2 and o_3 are output variables and o_2 will be assigned by the If command, hence we replace the "symbolic" dependency $\alpha(o_2) = \overline{O_2}$ by its "real" dependency $\Gamma(o_2) = O_2$. At the end of the If command, we do the join of the two environments obtained after the both branches, but in order to prevent cycles, we first replace the "symbolic"

dependencies by the corresponding "real" dependencies for each output variable that is assigned by the other branch.

As already stated above, our type system aims to capture the following non-interference condition: given a derivation $p \vdash_\alpha \Gamma\{c\}\Gamma'$, the final value of a variable x with final type $t \sqcup \alpha(X_O)$, should depend at most on the initial values of those variables y with initial types $\Gamma(y) \sqsubseteq t$ and on the final values of variables in X_O. Or otherwise said, executing a program c on two initial states σ and ρ such that $\sigma(y) = \rho(y)$ for all y with $\Gamma(y) \sqsubseteq t$ which ends with two final states σ' and ρ' such that $\sigma'(o) = \rho'(o)$ for all $o \in X_O$ will satisfy $\sigma'(x) = \rho'(x)$ for all x with $\Gamma'(x) \sqsubseteq t \sqcup \alpha(X_O)$. In order to prove the soundness of the typing system, we need a stronger invariant denoted $\mathcal{I}(t, \Gamma)$: intuitively, $(\sigma, \rho) \in \mathcal{I}(t, \Gamma)$ means that for each variable x and $A \subseteq X_O$, if $\sigma =_A \rho$ and $\Gamma(x) \sqsubseteq t \sqcup \alpha(A)$, then $\sigma(x) = \rho(x)$. Formally, given $t \in \mathcal{L}$ and $\Gamma : Var \mapsto \mathcal{L}$, we define $\mathcal{I}(t, \Gamma) \stackrel{def}{=} \bigcap_{A \subseteq X_O} =_{\Gamma, A, \alpha(A) \sqcup t}$. The following theorem states the soundness of our typing system.

Theorem 1. *Let us assume that $\mathcal{AC}(\Gamma)$ and $\forall o \in X_O, \alpha(o) \not\sqsubseteq t$. If $p \vdash_\alpha \Gamma\{c\}\Gamma'$ then $c : \mathcal{I}(t, \Gamma) \Longrightarrow \mathcal{I}(t, \Gamma')$.*

2.3 Soundness w.r.t. to output-sensitive non-interference

In this section we show how we can use the typing system in order to prove that a program c is output-sensitive noninterferent. Let $Var^e = Var \cup \{\overline{o} \mid o \in X_O\}$. Let us define $\mathcal{L} \stackrel{def}{=} \{\tau_A \mid A \subseteq Var^e\}$. We denote $\bot = \tau_\emptyset$ and $\top = \tau_{Var^e}$ and we consider the lattice $(\mathcal{L}, \bot, \top, \sqsubseteq)$ with $\tau_A \sqcup \tau_{A'} \stackrel{def}{=} \tau_{A \cup A'}$ and $\tau_A \sqsubseteq \tau_{A'}$ iff $A \subseteq A'$. The following Theorem is a consequence of the Definition 1 and Theorem 1.

Theorem 2. *Let $\mathcal{L}$ be the lattice described above. Let (Γ, α) be defined by $\Gamma(x) = \{\tau_x\}$, for all $x \in Var$ and $\alpha(o) = \{\tau_{\overline{o}}\}$, for all $o \in X_O$. If $\vdash_\alpha \Gamma\{c\}\Gamma'$ and for all $x_l \in X_L$, $\Gamma'(x_l) \sqsubseteq \Gamma(X_I) \sqcup \alpha(X_O)$, then c is (X_I, X_O, X_L)-secure.*

3 Output-sensitive constant-time

Following [1, 4], we consider two types of cache-based information leaks: 1) disclosures that happen when secret data determine which parts of the program are executed; 2) disclosures that arise when acces to memory is indexed by sensitive information. In order to model the latter category, we shall enrich the simple language from section 2.2 with *arrays*:

$$c ::= x := e \mid x[e_1] := e \mid \mathsf{skip} \mid c_1; c_2 \mid \mathsf{If}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \mid \mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}$$

To simplify notations, we assume that array indexes e_1 are basic expressions (not referring to arrays) and that X_O does not contain arrays. Moreover as in [4], a state or store σ maps array variables v and indices $i \in \mathbb{N}$ to values $\sigma(v, i)$. The labeled semantics of While programs are listed in Figure 4. In all rules, we

$$\frac{act \equiv \mathbf{r}(\sigma(\overrightarrow{f}))}{(x := e, \sigma) \xrightarrow{act} \sigma[(x,0) \mapsto \sigma(e)]} \qquad \frac{act \equiv \mathbf{w}(\sigma(e_1)) : \mathbf{r}(\sigma(\overrightarrow{f}))}{(x[e_1] := e, \sigma) \xrightarrow{act} \sigma[(x,\sigma(e_1)) \mapsto \sigma(e)]}$$

$$\frac{\sigma(e) \neq 1 \qquad act \equiv \mathbf{b}(\sigma(e)) : \mathbf{r}(\sigma(\overrightarrow{f}))}{(\mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}\ , \sigma) \xrightarrow{act} \sigma} \qquad \frac{\sigma(e) = 1 \qquad act \equiv \mathbf{b}(\sigma(e)) : \mathbf{r}(\sigma(\overrightarrow{f}))}{(\mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}\ , \sigma) \xrightarrow{act} (c; \mathsf{While}\ e\ \mathsf{Do}\ c\ \mathsf{oD}\ , \sigma)}$$

$$\frac{\sigma(e) = 1\ ?\ i = 1 :\ i = 2 \qquad act \equiv \mathbf{b}(\sigma(e)) : \mathbf{r}(\sigma(\overrightarrow{f}))}{(\mathsf{If}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ , \sigma) \xrightarrow{act} (c_i, \sigma)}$$

Fig. 4. Syntax and Labeled Operational semantics

$$\text{As1'}\ \frac{x \notin X_O}{p \vdash^{ct}_{\alpha} \Gamma\{x := e\}\Gamma[x \mapsto p \sqcup \Gamma[\alpha](fv(e))][x_l \mapsto \Gamma(x_l) \sqcup \Gamma[\alpha](fv(\overrightarrow{f}))]}$$

$$\text{As1"}\ \frac{x \notin X_O \qquad p_1 = (\Gamma[\alpha](fv(e_1), fv(e)) \qquad p_l = (\Gamma[\alpha](fv(e_1), fv(\overrightarrow{f}))}{p \vdash^{ct}_{\alpha} \Gamma\{x[e_1] := e\}\Gamma[x \mapsto p \sqcup \Gamma(x) \sqcup p_1][x_l \mapsto \Gamma(x_l) \sqcup p_l]}$$

$$\text{If}\ \frac{\begin{array}{cc} p' = (\Gamma[\alpha](fv(e)), \Gamma) \lhd_\alpha \mathbf{aff^O}(c_1; c_2) & p \sqcup p' \vdash^{ct}_{\alpha} \Gamma\{c_i\}\Gamma_i \\ p_l = (\Gamma[\alpha](fv(\overrightarrow{f})), \Gamma) \lhd_\alpha \mathbf{aff^O}(c_1; c_2) & \Gamma' = \Gamma_1 \lhd_\alpha \mathbf{aff^O}(c_2) \sqcup \Gamma_2 \lhd_\alpha \mathbf{aff^O}(c_1) \end{array}}{p \vdash^{ct}_{\alpha} \Gamma\{\mathsf{If}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ \}\Gamma'[x_l \mapsto \Gamma'(x_l) \sqcup p_l \sqcup p']}$$

Fig. 5. Typing Rules for Output Sensitive Constant Time (excerpts)

denote $\overrightarrow{f} = (f_i)_i$, where $x_i[f_i]$ are the indexed variables in e. The labels on the execution steps correspond to the information which is leaked to the environment ($\mathbf{r}()$ for a read access on memory, $\mathbf{w}()$ for a write access and $\mathbf{b}()$ for a branch operation). In the rules for (If) and (While) the valuations of branch conditions are leaked. Also, all indexes to program variables read and written at each statement are exposed. We give in Fig. 5 an excerpts of the new typing rules. As above, we denote $\overrightarrow{f} = (f_i)_i$, where $x_i[f_i]$ are the indexed variables in e. We add a fresh variable x_l, that is not used in programs, in order to capture the unintended leakage. Its type is always growing and it mirrors the information leaked by each command. In rule ($As1$") we take a conservative approach and we consider that the type of an array variable is the lub of all its cells. The information leaked by the assignment $x[e_1] := e$ is the index e_1 plus the set $\overrightarrow{f} = (f_i)_i$ of all indexes occurring in e. Moreover, the new type of the array variable x mirrors the fact that now the value of x depends also on the index e_1 and the right side e.

Definition 4. *An* **execution** *is a sequence of visible actions:* $\xrightarrow{a_1}\xrightarrow{a_2} \ldots \xrightarrow{a_n}$. *A program c is* **(X_I, X_O)-constant time** *when all its executions starting with X_I-equivalent stores that lead to finally X_O-equivalent stores, are identical.*

Following [4], given a set X of program variables, two stores σ and ρ are *X-equivalent* when $\sigma(x,i) = \rho(x,i)$ for all $x \in X$ and $i \in \mathbb{N}$. Two executions $\xrightarrow{a_1} \ldots \xrightarrow{a_n}$ and $\xrightarrow{b_1} \ldots \xrightarrow{b_m}$ are *identical* iff $n = m$ and $a_j = b_j$ for all $1 \leq j \leq n$. We can reduce the (X_I, X_O)-constant time security of a command

$\bullet$	$\omega(\bullet)$
$x := e$	$x_l := x_l : \mathbf{r}(\overrightarrow{f});\quad x := e$
$x[e_1] := e$	$x_l := x_l : \mathbf{w}(e_1) : \mathbf{r}(\overrightarrow{f});\quad x[e_1] := e$
$skip$	$skip$
$c_1; c_2$	$\omega(c_1); \omega(c_2)$
If e then c_1 else c_2 fi	$x_l := x_l : \mathbf{b}(e) : \mathbf{r}(\overrightarrow{f});\quad$ If e then $\omega(c_1)$ else $\omega(c_2)$ fi
While e Do c oD	$x_l := x_l : \mathbf{b}(e) : \mathbf{r}(\overrightarrow{f})$; While e Do $\omega(c); x_l := x_l : \mathbf{b}(e) : \mathbf{r}(\overrightarrow{f})$ oD

Fig. 6. Instrumentation for $\omega(\bullet)$

c to the $(X_I, X_O, \{x_l\})$-security (see section 2.3) of a corresponding command $\omega(c)$, obtained by adding a fresh variable x_l to the program variables $fv(c)$, and then adding recursively before each assignment and each boolean condition predicate, a new assignment to the leakage variable x_l that mirrors the leaked information. Let $:, \mathbf{b}(,)\mathbf{r}(,)\mathbf{w}()$ be some new abstract operators. The construction of the instrumentation $\omega(\bullet)$ is shown in Fig. 6. As above, we denote $\overrightarrow{f} = (f_i)_i$, where $x_i[f_i]$ are the indexed variables in e. Then, we extend, as in the rules $As1', Ass1$" from Fig 5, the typing system from section 2.2 to take into account the array variables. The following lemma holds.

Lemma 1. *Let c a command such that $x_l \notin fv(c)$, σ, σ' two stores , tr some execution trace and $[]$ the empty trace.*
1. $p \vdash^{ct}_{\alpha} \Gamma\{c\}\Gamma'$ iff $p \vdash_{\alpha} \Gamma\{\omega(c)\}\Gamma'$.
2. $(c, \sigma) \xrightarrow{tr}{}^{*} \sigma'$ iff $(\omega(c), \sigma[x_l \mapsto []]) \longrightarrow^{*} \sigma'[x_l \mapsto tr]$.

Now combining Theorem 2 and Lemma 1 we get the following Theorem.

Theorem 3. *Let $\mathcal{L}$ be the lattice defined in the section 2.3. Let (Γ, α) be defined by $\Gamma(x) = \{\tau_x\}$, for all $x \in Var$ and $\alpha(o) = \{\tau_{\overline{o}}\}$, for all $o \in X_O$ and $\Gamma(x_l) = \bot$. If $p \vdash^{ct}_{\alpha} \Gamma\{c\}\Gamma'$ and $\Gamma'(x_l) \sqsubseteq \Gamma(X_I) \sqcup \alpha(X_O)$, then c is (X_I, X_O)- constant time.*

4 Application to low-level code

We show in this section how the type system we proposed to express output-sensitive constant-time non-interference on the *While* language can be lifted to a low-level program representation like the LLVM byte code [21].

4.1 LLVM-IR

We consider a simplified LLVM-IR representation with four instructions: assignments from a temporary expression (register or immediate value) or from a memory location (load), writing to a memory location (store) and (un)conditional jump instructions. We assume that the program control flow is represented by a control-flow graph (CFG) $G = (\mathcal{B}, \rightarrow_E, b_{init})$ where $\mathcal{B}$ is the set of basic blocks,

$r \leftarrow op(Op, \overrightarrow{v})$	assign to r the result of Op applied to operands $\overrightarrow{v}$
$r \leftarrow load(v)$	load in r the value stored at address v
$store(v_1, v_2)$	store at address v_2 the value stored at address v_1
$cond(r, b_{then}, b_{else})$	branch to b_{then} if the value of r is true and to b_{false} otherwise
$goto\ b$	branch to b

Fig. 7. Syntax and informal semantics of simplified LLVM-IR

$\rightarrow_E$ the set of edges connecting the basic blocks, and $b_{init} \in \mathcal{B}$ the entry point. We denote by $Reach(b, b')$ the predicate indicating that there exists a path from b to b'. A program is a (partial) map from control points $(b, n) \in \mathcal{B} \times \mathbb{N}$ to instructions. Each basic block is terminated by a jump instruction. The memory model consists in a set of *registers* R and the memory M (including the execution stack). Val is the set of values and memory addresses. The informal semantics of our simplified LLVM-IR is given in Figure 7, where $r \in R$ and $v \in R \cup Val$.We consider an operational semantics where execution steps are labelled with leaking data, i.e., addresses of store and load operations and branching conditions.

4.2 Type system

For a CFG $G = (\mathcal{B}, \rightarrow_E, b_{init})$:

1. Function $dep : \mathcal{B} \rightarrow 2^{\mathcal{B}}$ associates to each basic block its set of "depending blocks", i.e., $b' \in dep(b)$ iff b' dominates b and there is no block b" between b' and b such that b" post-dominates b'. We recall that a node b_1 dominates (*resp.* post-dominates) a node b_2 iff every path from the entry node to b_2 goes through b_1 (*resp.* every path from b_2 to the ending node goes through b_1).

2. Partial function $br : \mathcal{B} \rightarrow R$ returns the "branching register", i.e., the register r used to compute the branching condition leading outside b (b is terminated by an instruction $cond(r, b_{then}, b_{else})$). Note that in LLVM branching registers are always *fresh* and assigned only once before to be used.

3. Function $PointsTo : (\mathcal{B} \times \mathbb{N}) \times Val \rightarrow 2^R$ returns the set of registers containing memory locations pointed to by a given address at a given control point. For example, for a given address v, $r \in PointsTo(b, n)(v)$ means that register r contains a memory address pointed to by v.

We define a type system to express output-sensitive constant-time property on LLVM-IR. The main differences from the rules at the source level is that the control-flow is explicitly given by the CFG. For lack of space we describe only the rule for the Store instruction (Figure 8). It updates the type of v_1 by adding the dependencies of all memory locations pointed to by $v2$. In addition, the type of the leakage variable x_l is also updated with the dependencies of all these memory locations lying in A_m (since these locations are read).

$$\text{St}\ \frac{\begin{array}{ccc} & A_m = PointsTo(b,n)(v_2) & \\ p(b,n) = store(v_1,v_2) & A_0 = A_m \cap X_0 & \Gamma_1 = (\Gamma,\alpha) \triangleleft A_0 \\ \tau_0 = \bigsqcup_{x \in br(dep(b))} \Gamma[\alpha](x) & \tau_1 = \Gamma_1[\alpha](v_2) \sqcup \tau_0 & \tau_2 = \Gamma_1[\alpha](v_1) \end{array}}{\vdash_\alpha (b,n) : \Gamma \Rightarrow \Gamma_1[x_l \to \Gamma_1(x_l) \sqcup \tau_1][v_{s_{\in A_m}} \to \Gamma(v_s) \sqcup \tau_2 \sqcup \tau_0]}$$

Fig. 8. store instruction

4.3 Well typed LLVM programs are output-sensitive constant-time

Definition 5. *An LLVM-IR program p is well typed with respect to an initial environment Γ_0 and final environment Γ' (written $\vdash_\alpha p : \Gamma_0 \Rightarrow \Gamma'$) , if there is a family of* well-defined *environments $\{(\Gamma)_{(b,n)} \mid (b,n) \in (\mathcal{B}, \mathbb{N})\}$, such that for all nodes (b,n) and all its successors (b',n'), there exists a type environment γ and $A \subseteq X_O$ such that $\vdash_\alpha (b,n) : \Gamma_{(b,n)} \Rightarrow \gamma$ and $(\gamma \lhd_\alpha A) \sqsubseteq \Gamma_{(b',n')}$.*

In the above definition the set A is mandatory in order to prevent dependency cycles between variables in X_O. The following Theorem shows the soundness of the typing system with respect to output-sensitive constant-time.

Theorem 4. *Let $\mathcal{L}$ be the lattice from the section 2.3. Let (Γ,α) be defined by $\Gamma(x) = \{\tau_x\}$, for all $x \in R \cup M$, $\alpha(o) = \{\tau_{\overline{o}}\}$, for all $o \in X_O$ and $\Gamma(x_l) = \bot$. If $\vdash_\alpha p : \Gamma \Rightarrow \Gamma'$ and $\Gamma'(x_l) \sqsubseteq \Gamma(X_I) \sqcup \alpha(X_O)$, then p is (X_I, X_O)- constant time.*

4.4 Implementation

We developed a prototype tool implementing the type system for LLVM programs. This type system consists in computing flow-sensitive dependency relations between program variables. Def. 5 provides the necessary conditions under which the obtained result is sound (Theorem 4). We give some technical indications regarding our implementation.

Output variables X_O are defined as function return values and global variables; we do not currently consider arrays nor pointers in X_O. Control dependencies cannot be deduced from the syntactic LLVM level, we need to explicitly compute the dominance relation between basic blocks of the CFG (the *dep* function). Def. 5 requires the construction of a set $A \subseteq X_O$ to update the environment produced at each control locations in order to avoid circular dependencies (when output variable are assigned in *alternative* execution paths). To identify the set of basic blocks belonging to such alternative execution paths leading to a given block, we use the notion of *Hammock regions* [15]. More precisely, we compute function $Reg : (\mathcal{B} \times \mathcal{B} \times (\to_E)) \to 2^{\mathcal{B}}$, returning the set of *Hammock regions* between a basic block b and its immediate dominator b' with respect to an incoming edge e_i of b. Thus, $Reg(b', b, (c,b))$ is the set of blocks belonging to CFG paths going from b' to b without reaching edge $e_i = (c,b)$: $Reg(b', b, (c,b)) = \{b_i \mid b' \to_E b_1 \cdots \to_E b_n \to_E b \wedge \forall i \in [1, n-1].\ \neg Reach(b_i, c)\}$.

Fix-point computations are implemented using Kildall's algorithm. To better handle real-life examples we are currently implementing the $PointsTo$ function, an inter-procedural analysis, and a more precise type analysis combining both over- and under-approximations of variable dependencies (see section 6).

5 Related Work

Information flow. There is a large number of papers on language-based security aiming to prevent undesired information flows using type systems (see [26]). An information-flow security type system statically ensures noninterference, i.e. that sensitive data may not flow directly or indirectly to public channels [30, 24, 29, 28]. The typing system presented in section 2.2 builds on ideas from Hunt and Sands' flow-sensitive static information-flow analysis [20].

As attractive as it is, noninterference is too strict to be useful in practice, as it prevents confidential data to have any influence on observable, public output: even a simple password checker function violates noninterference. Relaxed definitions of noninterference have been defined in order to support such intentional downward information flows [27]. Li and Zdancewic [22] proposed an expressive mechanism called *relaxed noninterference* for declassification policies that supports the extensional specification of secrets and their intended declassification. A declassification policy is a function that captures the precise information on a confidential value that can be *declassified*. For the password checker example, the following declassification policy $\lambda p.\lambda x.h(p) == x$, allows an equality comparison with the hash of password to be declassified (and made public), but disallows arbitrary declassifications such as revealing the password.

The problem of information-flow security has been studied also for low level languages. Barthe and Rezk [8, 9] provide a flow sensitive type system for a sequential bytecode language. As it is the case for most analyses, implicit flows are forbidden, and hence, modifications of parts of the environment with lower security type than the current context are not allowed. Genaim and Spoto present in [16] a compositional information flow analysis for full Java bytecode.

Information flow applied to detecting side-channel leakages. Information-flow analyses track the flow of information through the program but often ignore information flows through side channels. Side-channel attacks extract sensitive information about a program's state through its observable use of resources such as time or memory. Several approaches in language-based security use security type systems to detect timing side-channels [1, 18]. Agat [1] presents a type system sensitive to timing for a small While-language which includes a transformation which takes a program and transforms it into an equivalent program without timing leaks. Molnar et al [23] introduce the program counter model, which is equivalent to path non-interference, and give a program transformation for making programs secure in this model.

FlowTracker [25] allows to statically detect time-based side-channels in LLVM programs. Relying on the assumption that LLVM code is in SSA form, they compute control dependencies using a sparse analysis [13] without building the whole

Program Dependency Graph. Leakage at assembly-level is also considered in [6]. They propose a fine-grained information-flow analysis for checking that assembly programs generated by CompCert are constant-time. Moreover, they consider a stronger adversary which controls the scheduler and the cache.

All the above works do not consider publicly observable outputs. The work that is closest to ours is [4], where the authors develop a formal model for constant-time programming policies. The novelty of their approach is that it is distinguishing not only between public and private input values, but also between private and publicly observable output values. As they state, this distinction poses interesting technical and theoretical challenges. Moreover, constant-time implementations in cryptographic libraries like OpenSSL include optimizations for which paths and addresses can depend not only on public input values, but also on publicly observable output values. Considering only input values as non-secret information would thus incorrectly characterize those implementations as non-constant-time. They also develop a verification technique based on the self-composition based approach [7]. They reduce the constant time security of a program P to safety of a product program Q that simulates two parallel executions of P. The tool operates at the LLVM bytecode level. The obtained bytecode program is transformed into a product program which is verified by the Boogie verifier [5] and its SMT tool suite. Their approach is complete only if the public output is ignored. Otherwise, their construction relies on identifying the branches whose conditions can only be declared benign when public outputs are considered. For all such branches, the verifier needs to consider separate paths for the two simulated executions, rather than a single synchronized path and in the worst case this can deteriorate to an expensive product construction.

6 Conclusion and Perspectives

In this paper we proposed a static approach to check if a program is output-sensitive constant-time, i.e., if the leakage induced through branchings and/or memory accesses do not overcome the information produced by (regular) observable outputs. Our verification technique is based on a so-called output-sensitive non-interference property, allowing to compute the dependencies of a leakage variable from both the initial values of the program inputs and the final values of its outputs. We developed a type system on a high-level **While** language, and we proved its soundness. Then we lifted this type system to a basic LLVM-IR and we developed a prototype tool operating on this intermediate representation, showing the applicability of our technique.

This work could be continued in several directions. One limitation of our method arising in practice is that even if the two snippets $x_l = h; o = h$ and $o = h; x_l = o$ are equivalent, only the latter can be typed by our typing system. We are currently extending our approach by considering also an under-approximation $\beta(\bullet)$ of the dependencies between variables and using "symbolic dependencies" also for non-output variables. Then the safety condition from Theorem 2 can be improved to something like "$\exists V$ such that $(\Gamma'(x_l) \lhd_\alpha V) \sqsubseteq$

$(\Gamma(X_I) \lhd_\alpha V) \sqcup (\beta'(X_O) \lhd_\alpha V) \sqcup \alpha(X_O)$". In the above example, we would obtain $\Gamma'(x_l) = \alpha(h) = \beta'(o) \sqsubseteq \alpha(o) \sqcup \beta'(o)$, meaning that the unwanted maximal leakage $\Gamma'(x_l)$ is less than the minimal leakage $\beta'(o)$ due to the normal output. From the implementation point of view, further developments are needed in order to extend our prototype to a complete tool able to deal with real-life case studies. This may require to refine our notion of arrays and to take into account arrays and pointers as output variables. We could also consider applying a sparse analysis, as in FlowTracker [25]. It may happen that such a pure static analysis would be too strict, rejecting too much "correct" implementations. To solve this issue, a solution would be to combine it with the dynamic verification technique proposed in [4]. Thus, our analysis could be used to find automatically which branching conditions are benign in the output-sensitive sense, which could reduce the product construction of [4]. Finally, another interesting direction would be to adapt our work in the context of quantitative analysis for program leakage, like in [14].

References

1. Agat, J.: Transforming out timing leaks. In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 40–53. ACM (2000)
2. Al Fardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the tls and dtls record protocols. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 526–540. IEEE (2013)
3. Albrecht, M.R., Paterson, K.G.: Lucky microseconds: A timing attack on amazon's s2n implementation of tls. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 622–643. Springer (2016)
4. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 53–70. USENIX Association, Austin, TX (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida
5. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. vol. 5, pp. 364–387. Springer (2005)
6. Barthe, G., Betarte, G., Campo, J., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1267–1279. ACM (2014)
7. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE. pp. 100–114. IEEE (2004)
8. Barthe, G., Rezk, T.: Secure information flow for a sequential java virtual machine. In: TLDI'05: Types in Language Design and Implementation. Citeseer (2003)
9. Barthe, G., Rezk, T., Basu, A.: Security types preserving compilation. Computer Languages, Systems & Structures **33**(2), 35–59 (2007)
10. Bernstein, D., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. Progress in Cryptology–LATINCRYPT 2012 pp. 159–176 (2012)

11. Blazy, S., Pichardie, D., Trieu, A.: Verifying constant-time implementations by abstract interpretation. In: European Symposium on Research in Computer Security. pp. 260–277. Springer (2017)
12. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks **48**(5), 701–716 (2005)
13. Choi, J.D., Cytron, R., Ferrante, J.: Automatic construction of sparse data flow evaluation graphs. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 55–66. POPL '91, ACM (1991)
14. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. **18**(1), 4:1–4:32 (Jun 2015). https://doi.org/10.1145/2756550, http://doi.acm.org/10.1145/2756550
15. Ferrante, J., Ottenstein, K., Warren, J.: The program dependence graph and its use in optimization. TOPLAS **9**(3), 319–349 (1987)
16. Genaim, S., Spoto, F.: Information flow analysis for java bytecode. In: Verification, Model Checking, and Abstract Interpretation. pp. 346–362. Springer (2005)
17. Gullasch, D., Bangerter, E., Krenn, S.: Cache games–bringing access-based cache attacks on aes to practice. In: Security and Privacy (SP), 2011 IEEE Symposium on. pp. 490–505. IEEE (2011)
18. Hedin, D., Sands, D.: Timing aware information flow security for a javacard-like bytecode. Electronic Notes in Theoretical Computer Science **141**(1), 163–182 (2005)
19. Hunt, S., Sands, D.: Binding time analysis: A new perspective. In: In Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91). pp. 154–164. ACM Press (1991)
20. Hunt, S., Sands, D.: On flow-sensitive security types. In: ACM SIGPLAN Notices. vol. 41, pp. 79–90. ACM (2006)
21. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. CGO '04, IEEE Computer Society, Washington, DC, USA (2004)
22. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Proceedings of POPL. vol. 40, pp. 158–170. ACM (2005)
23. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: ICISC. vol. 3935, pp. 156–168. Springer (2005)
24. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 228–241. ACM (1999)
25. Rodrigues, B., Quintão Pereira, F.M., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 110–120. ACM (2016)
26. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications **21**(1), 5–19 (2003)
27. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. Journal of Computer Security **17**(5), 517–548 (2009)
28. Swamy, N., Chen, J., Chugh, R.: Enforcing stateful authorization and information flow policies in fine. In: ESOP. pp. 529–549. Springer (2010)
29. Vaughan, J.A., Zdancewic, S.: A cryptographic decentralized label model. In: Security and Privacy, 2007. SP'07. IEEE Symposium on. pp. 192–206. IEEE (2007)
30. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of computer security **4**(2-3), 167–187 (1996)