The Complexity of Quantitative Information Flow in Recursive Programs

Rohit Chadha[1] **and Michael Ummels**[2]

1 **University of Missouri, Columbia**
`chadhar@missouri.edu`
2 **Technische Universität Dresden**
`ummels@tcs.inf.tu-dresden.de`

Abstract

Information-theoretic measures based upon mutual information can be employed to quantify the information that an *execution* of a program reveals about its *secret inputs*. The *information leakage bounding problem* asks whether the information leaked by a program does not exceed a given threshold. We consider this problem for two scenarios: a) the *outputs* of the program are revealed, and b) the *timing* (measured in the number of execution steps) of the program is revealed. For both scenarios, we establish complexity results in the context of deterministic boolean programs, both for programs with and without recursion. In particular, we prove that for recursive programs the information leakage bounding problem is no harder than checking reachability.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.4.6 Security and Protection, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases quantitative information flow, recursive programs, program analysis, verification, computational complexity

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2012.534

1 Introduction

Ensuring that a program preserves confidentiality of its *secret inputs* is a fundamental problem in security. Typically, one desires that the execution of the program reveals absolutely *no* information about its secret inputs. This desired property is often modeled as *non-interference* [18, 29] — the *low-security observations* of the execution of a program should be independent of the *high-security inputs*. These observations could be *explicit* outputs of the program (e.g., the results of an election or whether a password is correct or not), or they could be *implicitly* extracted from its execution (such as timing information, cache size or power consumption).

In practice, however, non-interference is hard to achieve as it often clashes with functionality. An unanimous election, for example, reveals the votes of each voter. Consequently, alternative approaches that aim to quantify the amount of information leakage have been proposed in the literature [15, 19, 25, 31]. In these information-theoretic approaches, programs are viewed as *transformers* of random variables — they transform a random variable taking values from the set of inputs into a random variable taking values from the set of observations. Intuitively, the amount of information leaked by the program is the difference between the initial uncertainty and the uncertainty remaining in the high-security inputs given the observations from running the program; the formal definition relies on the notion of *mutual information*, which is based on the seminal work of Shannon [30].

32nd Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012).
Editors: D. D'Souza, J. Radhakrishnan, and K. Telikepalli; pp. 534–545
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Given the importance of this problem, several automated approaches have been proposed to compute the information leaked by the program. The techniques employed by these approaches range from model-checking [3, 24, 10, 11] and static analysis [12, 13, 14, 3] to statistical analysis [24, 9]. From a more theoretical viewpoint, the *complexity* of computing the amount of leakage was only considered recently [34, 33, 35, 7]. More precisely, [33, 35, 7] consider the complexity of the *information leakage bounding problem*: given a program P, a distribution μ on the set of inputs, and a rational number q, check if the information leaked by the program (denoted by $\mathrm{SE}_\mu(P)$) does not exceed q.

In [33, 35], the program P is described in a simple non-recursive deterministic imperative language with boolean variables, assignments, conditionals and loops, and the inputs are assumed to be uniformly distributed. They show that the information leakage bounding problem is PP-hard for the loop-free fragment.[1] For the whole language with loops, the problem is shown to be PSPACE-hard. However, no upper bounds are given in [33, 35]. An EXPSPACE upper bound can be derived from the work of Černý et al. [7], where the information leakage bounding problem is shown to be PSPACE-complete under the assumption that the program is represented explicitly as a nondeterministic transition system and the input distribution is given explicitly. In our setting, an exponential blow-up occurs because the translation from a boolean program to a nondeterministic transition system is exponential.

Contributions. Our first contribution is an upper bound for loop-free boolean programs when the number of output variables is logarithmic in the size of the program.[2] We show that in this case the information leakage bounding problem for uniformly distributed inputs lies in the fourth level of the *counting hierarchy* (more precisely in $\mathsf{P}^{\mathsf{CH}_3}$). The whole counting hierarchy is contained in PSPACE. The main challenge in establishing the upper bound is that we have to solve inequations that involve logarithms (because of the definition of mutual information). In order to overcome this challenge, we resort to recent breakthroughs in arithmetic circuit complexity [2]. We then employ similar techniques to establish PSPACE-completeness for boolean programs with loops (but no recursion) under the same assumption that the number of output variables is logarithmic in the size of program. Hence, our upper bound is a substantial improvement over the previous EXPSPACE upper bound.

We subsequently turn our attention to boolean programs with recursion. We show that both the problem of checking non-interference as well as the information leakage bounding problem is EXPTIME-complete. For the upper bound, we observe that a recursive boolean program can be represented as an *exponential-size* deterministic pushdown system [26, 28] (the pushdown system is of size linear in the length of the program but exponential in the number of variables). We can then use the fact that control state reachability in pushdown systems is polynomial-time decidable [5] and thus compute the outputs of the program on any given input. A careful analysis of the expression computing information leakage then gives us the desired upper bound. We make no assumptions on the number of variables in this case, and hence this also gives an EXPTIME upper bound for general non-recursive programs, which is better than the EXPSPACE upper bound that can be derived from [7].

In the second part of this paper, we consider the case when the attacker can observe the timing behavior of an execution of a program. We abstract the timing behavior of the program by the "length" of the computation of the pushdown system corresponding to the program. One could alternatively use the number of procedure calls or the number

---

[1] Recall that PP is the class of decision problems decidable by a probabilistic polynomial-time Turing machine with acceptance probability $\geq 1/2$.

[2] If one also allows *low-level* security input variables, their number must also be bounded.

of loop executions as an abstraction of timing, but our results would not change in that case. For non-recursive terminating programs, the execution time can easily be measured "inside the program" by a binary counter, so bounding the information leaked by timing is no harder than the problem of bounding the information leaked by observing the outputs. The same idea does not work for recursive programs because the running time of a recursive program could be doubly exponential. Nevertheless, we show that the problem of bounding the information leaked by the timing behavior of a recursive boolean program on uniformly distributed inputs is also EXPTIME-complete by showing that the execution time for recursive terminating programs can still be computed in exponential time.

Related work. The complexity of quantitative information flow security for boolean programs was first tackled by Yasuoka and Terauchi [34], where the complexity of the information leakage comparison problem is studied: this problem asks which of two programs leaks more information. They also show that the problem of checking non-interference for loop-free programs is coNP-complete. The complexity of bounding information leakage was first studied by the same authors in [33], where the problem was shown to be PP-hard for loop-free non-recursive programs. In [35], the same authors prove that deciding non-interference for non-recursive programs *with loops* is PSPACE-complete. However, none of these papers contains an upper bound for the problem of bounding Shannon-entropy based information leakage, not even for restricted programs. Thus, our results obtained by restricting the number of output variables are novel. Only for the related notions of *min-entropy* and *guessing entropy*, a PSPACE upper and lower bound for non-recursive programs was established in [35].

A more general setting has been considered by van der Meyden and Zhang [32] as well as Černý et al. [7], where programs are represented abstractly as nondeterministic transition systems. In this setting, van der Meyden and Zhang established PSPACE-completeness for noninterference, and Černý et al. extended this result to the information-leakage bounding problem (wrt. Shannon entropy). However, as they assume an explicit-state description and the translation of a boolean program into an equivalent explicit-state description causes an exponential blowup, their results only give an EXPSPACE-upper bound for boolean programs (without recursion). None of these works consider recursive programs or the problem of bounding the information leakage caused by timing information. We establish EXPTIME-completeness for both problems, and also obtain better bounds for non-recursive programs.

Several timing attacks are known in literature. For example, [6] shows a practical timing attack against OpenSSL, which allows extraction of a private RSA key. The attack exploits the fact that the multiplication in OpenSSL is carried out by the Karatsuba routine [21], which is a recursive algorithm. Several approaches have been proposed in the literature to counteract timing leaks. Type systems, for example, are used to detect information leakage from timing [20], while [1, 23, 27, 4] provide countermeasures to combat information leakage from timing. None of these works have considered complexity questions, though.

Note. Due to space constraints, most proofs are only sketched or omitted entirely. For details, see the full version of this paper [8].

2 Preliminaries

All logarithms are to the base 2. As is standard, we define $0 \log 0 = 0$. We assume that the reader is familiar with probability distributions and (discrete) random variables. Given a function $f\colon A \to B$ and $b \in B$, the set $\{a \in A \mid f(a) = b\}$ is denoted by $f^{-1}(b)$. Finally, we denote by 2^A the set of all (total) functions from A to the set $\{\top, \bot\}$.

Straight-line programs and the counting hierarchy. A (division-free) *straight-line program* is a finite list of instructions of the form $x \leftarrow c$ or $x \leftarrow y \odot z$, where $c \in \{0, 1\}$, $\odot \in \{+, -, \cdot\}$ and x, y, z are taken from a countable set of variables. Such a program is *closed* if all variables that appear on the right-hand side of an instruction also appear on the left-hand side of a preceding instruction. Hence, a closed straight-line program represents an integer, namely the value of the last variable that is assigned to. The problem PosSLP is to decide, given a closed straight-line program, whether the corresponding integer is > 0.

The counting hierarchy consists of the classes CH_i where $\mathsf{CH}_0 = \mathsf{P}$ and $\mathsf{CH}_{i+1} = \mathsf{PP}^{\mathsf{CH}_i}$ for all $i \in \mathbb{N}$. Allender et al. [2] recently showed that the Problem PosSLP belongs to the complexity class $\mathsf{P}^{\mathsf{CH}_3}$ and thus to the fourth-level of the counting hierarchy. Since the counting hierarchy is contained in PSPACE, we know in particular that PosSLP is decidable in polynomial space.

Pushdown Systems. The operational semantics of recursive programs are given by pushdown systems. Formally a pushdown system (PDS) $\mathcal{P}$ is a tuple (Q, Γ, δ) where Q is a finite set of *control states*, Γ is a finite *stack alphabet*, and $\delta = \delta_{\text{int}} \cup \delta_{\text{cll}} \cup \delta_{\text{rtn}}$ is a finite set of *transitions* s.t. $\delta_{\text{int}} \subseteq Q \times Q$, $\delta_{\text{cll}} \subseteq Q \times Q \times \Gamma$, and $\delta_{\text{rtn}} \subseteq Q \times \Gamma \times Q$.

A labeled transition system $(\text{Labels}, \text{Conf}_{\mathcal{P}}, \rightarrow_{\mathcal{P}})$ defines the semantics of a PDS $\mathcal{P}$. The set Labels of labels is $\{\text{int}, \text{cll}, \text{rtn}\}$. The set $\text{Conf}_{\mathcal{P}}$ of configurations is $Q \times \Gamma^*$. The word $w \in \Gamma^*$ in a configuration (q, w) models the contents of the stack; the empty word ε denotes the empty stack. The transition relation $\rightarrow_{\mathcal{P}}$ is defined as follows: $(q, w) \xrightarrow{\text{int}}_{\mathcal{P}} (q', w)$ if $(q, q') \in \delta_{\text{int}}$; $(q, w) \xrightarrow{\text{cll}}_{\mathcal{P}} (q', wa)$ if $(q, q', a) \in \delta_{\text{cll}}$ and $(q, wa) \xrightarrow{\text{rtn}}_{\mathcal{P}} (q', w)$ if $(q, a, q') \in \delta_{\text{rtn}}$.

We omit the subscript $\mathcal{P}$ if it is clear from the context. Since we consider only deterministic programs, we are mainly interested in *deterministic* PDS: $\mathcal{P}$ is deterministic if for each s in $\text{Conf}_{\mathcal{P}}$ there is *at most one* $\lambda \in \text{Labels}$ and *at most one* $s' \in \text{Conf}_{\mathcal{P}}$ with $s \xrightarrow{\lambda}_{\mathcal{P}} s'$.

Given a configuration $c = (q, w)$ of a PDS $\mathcal{P}$, we say that $\text{state}(c) = q, \text{stack}(c) = w$ and $\text{height}(c) = |w|$, the length of w. A *computation* of $\mathcal{P}$ is a sequence $c_0 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_m} c_m$. A transition $c_i \xrightarrow{\text{cll}} c_{i+1}$ is a *procedure call*. Similarly, we define *procedure returns* and *internal actions*. We say that a procedure return $c_j \xrightarrow{\text{rtn}} c_{j+1}$ *matches* a procedure call $c_i \xrightarrow{\text{cll}} c_{i+1}$ iff $i < j$, $\text{height}(c_{i+1}) = \text{height}(c_j)$ and $\text{height}(c_{i+1}) \leq \text{height}(c_k)$ for all $i < k < j$. Finally, we say that $c \overset{m}{\Rightarrow}_{\mathcal{P}} c'$ if there exists a computation $c_0 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_m} c_m$ of $\mathcal{P}$ with $c_0 = c$ and $c_m = c'$, and we write $c \Rightarrow_{\mathcal{P}} c'$ if $c \overset{m}{\Rightarrow}_{\mathcal{P}} c'$ for some $m \in \mathbb{N}$. The following is proved in [5].

▶ **Theorem 1.** *There are polynomial-time algorithms that, given a PDS $\mathcal{P}$, output the set $\{(q, q') \mid (q, \varepsilon) \Rightarrow_{\mathcal{P}} (q', \varepsilon)\}$ and the set $\{(q, q') \mid \exists w \in \Gamma^* \ (q, \varepsilon) \Rightarrow_{\mathcal{P}} (q', w)\}$, respectively.*

Programs. Due to space constraints, we cannot present the syntax of recursive boolean programs in detail. Here we just highlight the main features. The inputs of our programs are partitioned into two sets, one containing *high-security variables* and one containing *low-security variables*. Additionally, our programs may have some local variables as well as outputs. The outputs are assumed to be of *low security*. Note that high-security outputs, i.e., outputs that are not visible to an observer, can easily be modeled using local variables.

We only give an informal description of the semantics of programs, which is *call-by-value*. A recursive boolean program can be represented as a deterministic pushdown system [26, 28] of exponential size (linear in the length of the program, but exponential in the number of variables). The states of the pushdown system keep track of the current statement and the values of all variables in the "current scope"; the pushdown stack keeps track of the procedure calls. Whenever a procedure is called, the pushdown system pushes the position of the call and the values of the variables onto the stack, transitions into the called procedure, and sets all variables that are local to this procedure to $\perp$. Upon returning from the procedure call, the contents from the stack is popped and the variables are reset properly, i.e., the outputs of the returning procedure are set, and the variables that were local to the procedure are reset to their original values using the information from the stack. Since P is deterministic, the corresponding pushdown system is also deterministic.

The computation of the program P on high inputs $\bar{h}_0$ and low inputs $\bar{l}_0$ can now be defined as the computation of the pushdown system corresponding to P starting from the configuration with the empty stack and with the control state corresponding to the first statement of P, the input variables set to $\bar{h}_0, \bar{l}_0$, and the local and output variables set to $\perp$. The program P *terminates* on inputs $\bar{h}_0, \bar{l}_0$ if this computation reaches the configuration with the control state corresponding to the last statement of the program (in that case, the stack will be empty). If P terminates, we define the output of P to be the values of the output variables upon termination. Hence, P can be seen as a partial function $F_P\colon 2^{\bar{h}} \times 2^{\bar{l}} \to 2^{\bar{o}}$.

Henceforth, the program P is always assumed to be terminating. One could possibly model non-termination as an explicit observation; and our complexity results will not change in that case. This is because nontermination on an input can be decided for while programs in PSPACE and for recursive boolean programs in EXPTIME.

Quantifying information leakage. Let $\mathcal{X}$ be a discrete random variable with values taken from a finite set X. If μ is the probability distribution of $\mathcal{X}$, the *Shannon entropy* of μ, written $\mathrm{H}_\mu(\mathcal{X})$, is defined as

$$\mathrm{H}_\mu(\mathcal{X}) = -\sum_{x \in X} \mu(\mathcal{X} = x) \cdot \log \mu(\mathcal{X} = x).$$

If $\mathcal{X}$ and $\mathcal{Y}$ are discrete random variables taking values from finite sets X and Y with joint probability distribution μ, the *conditional entropy* of $\mathcal{X}$ given $\mathcal{Y}$, written $\mathrm{H}_\mu(\mathcal{X} \mid \mathcal{Y})$, is defined as

$$\mathrm{H}_\mu(\mathcal{X} \mid \mathcal{Y}) = \sum_{y \in Y} \mu(\mathcal{Y} = y) \cdot \mathrm{H}_\mu(\mathcal{X} \mid \mathcal{Y} = y),$$

where

$$\mathrm{H}_\mu(\mathcal{X} \mid \mathcal{Y} = y) = -\sum_{x \in X} \mu(\mathcal{X} = x \mid \mathcal{Y} = y) \cdot \log \mu(\mathcal{X} = x \mid \mathcal{Y} = y).$$

If $\mathcal{X}, \mathcal{Y}$ and $\mathcal{Z}$ are discrete random variables taking values from finite sets X, Y and Z with joint probability distribution μ, then the *joint conditional entropy* of $\mathcal{X}, \mathcal{Y}$ given $\mathcal{Z}$, written $\mathrm{H}_\mu(\mathcal{X}, \mathcal{Y} \mid \mathcal{Z})$ is the entropy of the random variable $(\mathcal{X}, \mathcal{Y})$ given $\mathcal{Z}$. Similarly, the conditional entropy of $\mathcal{X}$ given $\mathcal{Y}$ and $\mathcal{Z}$ is the entropy of $\mathcal{X}$ given $(\mathcal{Y}, \mathcal{Z})$.

If $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are discrete random variables taking values from finite sets X, Y and Z respectively with joint probability distribution μ, then the *conditional mutual information* of $\mathcal{X}$ and $\mathcal{Y}$ given $\mathcal{Z}$, written $\mathrm{I}_\mu(\mathcal{X}; \mathcal{Y} \mid \mathcal{Z})$, is defined as

$$\mathrm{I}_\mu(\mathcal{X}; \mathcal{Y} \mid \mathcal{Z}) = \mathrm{H}_\mu(\mathcal{X} \mid \mathcal{Z}) - \mathrm{H}_\mu(\mathcal{X} \mid \mathcal{Y}, \mathcal{Z}).$$

We are interested in measuring the information leaked by a program. Following [15, 19, 25], we use conditional mutual information to quantify this information. As described above, we can view programs as functions that take two kinds of inputs: a *high-security* (high) input from a finite set H and a *low-security* (low) input from a finite set L. Let $\mathcal{H}$ and $\mathcal{L}$ be random variables taking values from H and L, respectively, with joint distribution μ.

Moreover, let O be a finite set and $F\colon H \times L \to O$ be a function. We extend μ to a joint probability distribution on $\mathcal{H}$, $\mathcal{L}$ and $\mathcal{O}$ such that

$$\mu(\mathcal{O} = o \mid \mathcal{H} = h, \mathcal{L} = l) = \begin{cases} 1 & \text{if } F(h,l) = o \\ 0 & \text{otherwise} \end{cases}$$

The *information leaked by the function F* is then

$$\mathrm{SE}_\mu(F) := \mathrm{I}_\mu(\mathcal{H}; \mathcal{O} \mid \mathcal{L}).$$

We are mainly interested in the case where μ is the uniform distribution on $H \times L$, and we define $\mathrm{SE}_\mathsf{U}(F) := \mathrm{SE}_\mu(F)$ in this case.

A function $F\colon H \times L \to O$ is *non-interferent* if $F(h,l) = F(h',l)$ for all $h, h' \in H$ and $l \in L$, and *interferent* otherwise. Note that a function $F\colon H \times L \to O$ is non-interferent iff $\mathrm{SE}_\mathsf{U}(F) = 0$ for all distributions μ.

Sometimes, we have only high inputs, i.e., F is a function from H to O. In that case, the information leaked by the function F is just $\mathrm{SE}_\mu(F) = \mathrm{I}_\mu(\mathcal{H}; \mathcal{O})$. The following lemma allows us to trade low inputs for high inputs and outputs.

▶ **Lemma 2.** *Let H, L, O be finite sets, $F\colon H \times L \to O$, and let $\mathcal{H}$ and $\mathcal{L}$ be random variables taking values in H and L with joint probability distribution μ. Consider the function $G\colon (H \times L) \to (O \times L)$ defined by $G(h,l) = (F(h,l), l)$. Then $\mathrm{SE}_\mu(G) = \mathrm{SE}_\mu(F) + \mathrm{H}_\mu(\mathcal{L})$.*

The following theorem is proved in [3, 22].

▶ **Theorem 3.** *Let H and O be finite sets, and let $F\colon H \to O$ be a function. Then*

$$\mathrm{SE}_\mathsf{U}(F) = \log |H| - \frac{1}{|H|} \sum_{o \in O} |F^{-1}(o)| \log |F^{-1}(o)|.$$

The information leakage bounding problem. As discussed above, a program P with high input variables $\bar{h}$, low input variables $\bar{l}$ and output variables $\bar{o}$ can be seen as a function $F_P\colon 2^{\bar{h}} \times 2^{\bar{l}} \to 2^{\bar{o}}$. Now, the information leakage bounding problem asks, given a program P and a rational number $q \geq 0$, whether the information leaked by F_P does not exceed q, i.e. whether $\mathrm{SE}_\mathsf{U}(F_P) \leq q$. In the rest of the paper, we will identify P with the function F_P.

3 Complexity of information leakage

Loop-free programs. We start by discussing our results for loop-free programs. Given numbers $a_1, \ldots, a_k \in \mathbb{N}$, we define $\sigma(a_1, \ldots, a_k) = \sum_{i=1}^{k} a_i \log a_i$. Note that if $F\colon H \to O$ is a function, and $a_1, \ldots, a_k$ is a permutation of $\{|F^{-1}(o)| \mid o \in O\}$, then $\mathrm{SE}_\mathsf{U}(F) = \log|H| - \sigma(a_1, \ldots, a_k)/|H|$, according to Theorem 3.

▶ **Lemma 4.** *Given $a_1, \ldots, a_k \in \mathbb{N}$ and $q \in \mathbb{Q}$, deciding whether $\sigma(a_1, \ldots, a_k) < q$ reduces to* PosSLP *in polynomial time.*

Proof. In order to prove the lemma, we show that, given $a_1, \ldots, a_k, q$, one can construct a (division-free) straight-line program S in polynomial time such that $\sigma(a_1, \ldots, a_k) < q$ iff $S \in$ PosSLP. Since $\sigma(a_1, \ldots, a_k)$ is always nonnegative, we can assume that $q > 0$. Let $q = r/s$, where both $r, s \in \mathbb{N} \setminus \{0\}$. Hence, $\sigma(a_1 \ldots, a_k) < q$ iff $s \cdot \sigma(a_1 \ldots, a_k) < r$. Using the fact that $\log a + \log b = \log ab$ and $a \log b = \log b^a$, we have

$$s \cdot \sigma(a_1, \ldots, a_k) = \log \prod_{i=1}^{k} a_i^{a_i s}.$$

Applying an exponentiation on both sides, we get that

$$\sigma(a_1 \ldots, a_k) < q \iff \prod_{i=1}^{k} a_i^{a_i s} < 2^r \iff 0 < 2^r - \prod_{i=1}^{k} a_i^{a_i s}.$$

Now, using repeated squaring, we can write a straight-line program of size $\mathrm{O}(\log r + \log s + \sum_{i=1}^{k} \log a_i)$ representing the number on the right-hand side, establishing our reduction. ◀

We can now show that the information leakage bounding problem for loop-free programs lies inside the counting hierarchy, provided the number of possible low inputs and outputs is only logarithmic in the number of high inputs.

▶ **Theorem 5.** *Given a loop-free program P with $|\bar{o}|+|\bar{l}| = \mathrm{O}(\log |\bar{h}|)$ and a rational number q, deciding whether $\mathrm{SE}_{\mathsf{U}}(P) \leq q$ can be done in $\mathsf{P}^{\mathsf{CH}_3}$.*

Proof. First, observe that if P has k low input variables and $\mathcal{L}$ is the distribution induced by U on low inputs, then $\mathrm{H}_{\mathsf{U}}(\mathcal{L}) = k$. Using this observation and Lemma 2, it suffices to consider the case when P has only high inputs. Moreover, since $\mathsf{P}^{\mathsf{CH}_3}$ is closed under complementation, it suffices to show that we can decide whether $\mathrm{SE}_{\mathsf{U}}(P) > q$ in $\mathsf{P}^{\mathsf{CH}_3}$.

Let $|\bar{h}| = m$ and denote by H and O the set of possible inputs and outputs, respectively. Note that $|H| = 2^m$ and $|O| = \mathrm{O}(m^d)$ for some $d \in \mathbb{N}$ (since $|\bar{o}| = \mathrm{O}(\log m)$). Let $O = \{\bar{o}_1, \ldots, \bar{o}_k\}$ and for each $i = 1, \ldots, k$ set $a_i = |P^{-1}(\bar{o}_i)|$. Now, by Theorem 3, we have

$$\mathrm{SE}_{\mathsf{U}}(P) = m - 2^{-m} \cdot \sigma(a_1, \ldots, a_k)$$

and therefore

$$\mathrm{SE}_{\mathsf{U}}(P) > q \iff \sigma(a_1, \ldots, a_k) < 2^m(m-q)$$

Note that all the numbers a_i as well as $2^m(m-q)$ are of size polynomial in the size of P and the size of q. Hence, given $a_1, \ldots, a_k$, we can apply Lemma 4 and compute (in polynomial time) a straight-line program S such that $S \in \mathrm{PosSLP}$ iff $\mathrm{SE}_{\mathsf{U}}(P) > q$.

Since PosSLP is in $\mathsf{P}^{\mathsf{CH}_3}$ [2], we are done if we can show that the numbers $a_1, \ldots, a_k$ can be computed by a polynomial-time algorithm with an oracle for CH_3. In fact, we show that these numbers can be computed in $\#\mathsf{P}$; since $\#\mathsf{P} \subseteq \mathsf{P}^{\mathsf{PP}}$, this will conclude the proof. Given an output $\bar{o}_i \in O$, the *weakest precondition semantics* gives us a Boolean formula $\varphi_i(\bar{h})$, which can be computed in polynomial time [17], such that an assignment $\alpha \in 2^{\bar{h}}$ satisfies φ_i iff $P(\alpha) = \bar{o}_i$. Hence, $a_i = |\{\alpha \mid \alpha \models \varphi_i\}|$. Since the problem of computing the number of satisfying assignments for a given Boolean formula is in $\#\mathsf{P}$, we are done. ◀

While programs. Non-interference for while programs is shown to be PSPACE-complete in [35]. Indeed, it can be shown to be PSPACE-hard even for programs that have only one high input variable, no low input variables, and one output variable. We show that the upper bound extends to the information leakage bounding problem, provided the number of possible low inputs and outputs is only logarithmic in the number of high inputs.

▶ **Theorem 6.** *Given a while program P with $|\bar{o}| + |\bar{l}| = \mathrm{O}(\log |\bar{h}|)$ and a rational number q, deciding whether $\mathrm{SE}_{\mathsf{U}}(P) \leq q$ is PSPACE-complete.*

Proof. We prove PSPACE hardness for the special case of one high input, no low inputs, and one output in [8]. The proof for containment in PSPACE is almost identical to the proof of Theorem 5. The only difference is that we cannot transform a while program into an equivalent Boolean formula in polynomial time (or reachability for Boolean programs would be in NP). Instead, we just "run" the given program P on every possible input in order to compute the numbers $a_i = |P^{-1}(\bar{o}_i)|$, which can be done in polynomial space. Since the counting hierarchy is contained in PSPACE, this gives a polynomial-space algorithm. ◀

Recursive Programs. Deciding non-interference becomes EXPTIME-hard if we allow procedure calls, i.e., at least as hard as deciding reachability for recursive programs.

▶ **Theorem 7.** *Deciding non-interference for recursive programs with one high input, no low inputs, and one output is* EXPTIME*-hard.*

As a corollary, we get that the information leakage bounding problem for recursive programs is also EXPTIME-hard, even when the number of inputs and outputs is restricted. We now show that the information leakage bounding problem is indeed no harder than the reachability problem, i.e. is in EXPTIME. As opposed to our PSPACE upper bound for while programs, we will have no restriction on the number of inputs or on the number of outputs. In particular, the EXPTIME upper bound also applies to arbitrary while programs.

▶ **Theorem 8.** *The information leakage bounding problem is* EXPTIME*-complete for recursive programs.*

Proof. EXPTIME-hardness follows from Theorem 7. For the upper bound, as in the proof of Theorem 5, we can assume that P has m high inputs, no low inputs and n outputs, and that $0 \leq q < m$. Let H be the set of possible inputs to P and $O = \{\bar{o}_1, \ldots, \bar{o}_k\}$ the set of possible outputs. Hence, $|H| = 2^m$ and $k = |O| \leq 2^n$. As shown in the proof of Theorem 5, we have $\mathrm{SE}_{\mathsf{U}}(P) \leq q$ iff $\sigma(a_1, \ldots, a_k) \geq 2^m(m-q)$, where $a_i := |P^{-1}(\bar{o}_i)|$. Let $2^m(m-q) = r/s$, where $r, s \in \mathbb{N}$ (such numbers can be computed easily from P and q). Now, as in the proof of Lemma 4, we have

$$\mathrm{SE}_{\mathsf{U}}(P) \leq q \iff \log \prod_{i=1}^{k} a_i^{a_i s} \geq r.$$

Note that we have no restriction on the number of outputs. Hence, unlike in the proof of Theorem 5, we cannot appeal to Lemma 4. However, observe that $\sum_{i=1}^{k} a_i = 2^m$. Hence, by replacing the powers by products, we can write $p := \prod_{i=1}^{k} a_i^{a_i s}$ as a product of $2^m \cdot s$ natural numbers each of (binary) size at most m. The product of $2^m \cdot s$ natural numbers each of size at most m can be computed in $2^{\mathrm{O}(m \log s)}$ time and is of size $2^{\mathrm{O}(m \log s)}$. Now note that $\log p \geq r$ iff the integral part of the left-hand side is $\geq r$ (since the right-hand side is an integer), but the integral part of $\log p$ is just the length of the binary representation of p, which we have just computed.

To establish the EXPTIME upper bound, it remains to be shown that the numbers $a_i = |P^{-1}(\bar{o}_i)|$ can be computed in exponential time. This can be done by first computing the pushdown system corresponding to P, which is of size exponential in the size of P, and then invoking Theorem 1 to compute the set $\{(\bar{h}_0, F(\bar{h}_0)) \mid \bar{h}_0 \text{ is a high input}\}$. ◀

▶ Remark. The algorithm in the proof of Theorem 8 runs in time polynomial in the length of the program and exponential in the number of variables.

4 Information leakage from timing behavior

Let us now consider the question of estimating the information leaked by a program by its "timing behavior". We shall use the "number of steps" taken by a program as an abstraction of its timing behavior. Given a program P, with high input variables $\bar{h}$ and low input variables $\bar{l}$, let $\mathrm{Steps}_P : 2^{\bar{h}} \times 2^{\bar{l}} \to \mathbb{N}$ be the function such that $\mathrm{Steps}_P(\bar{h}_0, \bar{l}_0)$ is the number of steps in the computation of $P(\bar{h}_0, \bar{l}_0)$. More precisely, this number is the number of steps in the corresponding computation of the pushdown system realizing the program P.

▶ **Definition 9.** A program P is *timing non-interferent* if the function Steps_P is non-interferent. Furthermore, if μ is the distribution on inputs to P, then $\mathrm{SE}_\mu(\mathrm{Steps}_P)$ is the *information leaked by the timing behavior* of P.

While programs. A terminating while program takes at most $\ell \cdot 2^n$ steps, where ℓ is the number of statements in the program and n is the total number of variables of the program (input, output and local). Hence, the running time of the program can be represented as a natural number whose (binary) size is polynomial in the size of program. Thus, we can easily modify the upper bound proof for deciding non-interference in while programs to the case of deciding timing non-interference in while programs. The lower bound proof for deciding non-interference in while programs can also be easily modified to give a lower bound on timing non-interference of while programs.

▶ **Lemma 10.** *Deciding timing non-interference for while programs with one high input, no low inputs and no outputs is* PSPACE*-hard. Deciding whether a while program is timing non-interferent can be done in* PSPACE*.*

Recursive programs. As in the case of while programs, the lower bound for deciding timing non-interference for recursive programs is a modification of the proof for Theorem 7.

▶ **Lemma 11.** *Deciding timing non-interference for recursive programs with one high input, no low inputs and no outputs is* EXPTIME*-hard.*

The upper bound proofs for bounding information leakage are more involved. The presence of recursion (i.e., the stack) implies that the length of the computation is no longer bounded by $\ell \cdot 2^n$ as in the case of while programs. Indeed, the length of a computation can be as high as doubly exponential, and the upper bound proof will depend on the ability to compute the length of a computation in exponential time. (Note that the length of a computation can be represented as an exponential-size number). In order to demonstrate this fact, we will establish some facts about deterministic pushdown systems.

Given a deterministic PDS $\mathcal{P}$, we say that a computation $c_0 \xrightarrow{\lambda_1} c_1 \cdots \xrightarrow{\lambda_m} c_m$ of $\mathcal{P}$ is *terminating* if there is no transition out of c_m. A state $q \in Q$ is a *good state* if there exists a terminating computation $c_0 \xrightarrow{\lambda_1} c_1 \cdots \xrightarrow{\lambda_m} c_m$ with $c_0 = (q, \varepsilon)$. We first establish that that the length of a computation from a good state is of at most exponential length.

▶ **Lemma 12.** *Let $\mathcal{P} = (Q, \Gamma, \delta)$ be a deterministic PDS and $q \in Q$ a good state. If there exists a configuration c with $(q, \varepsilon) \overset{m}{\Rightarrow}_{\mathcal{P}} c$, then $m \leq |Q| \cdot |\Gamma|^{|Q|+1}$.*

We now show that, even though the length of a computation of a deterministic pushdown system can be exponential, the length of the computation from a configuration (q, ε) to (q', ε) can be computed in polynomial time. This is proved by modifying the “summaries construction” algorithm used to decide reachability in pushdown systems [5]. We recall salient points of this algorithm before we prove the desired theorem.

The “summaries construction” algorithm proceeds iteratively, building an edge-labeled graph on the states of a pushdown system $\mathcal{P}$. At each step of the algorithm, edges are added and the algorithm terminates when a fixed point is reached. The set of labels on the edges is $\Gamma \cup \{\varepsilon\}$. Intuitively, the edge $q \overset{a}{\rightsquigarrow} q'$ means that there is a valid computation $(q, \varepsilon) \Rightarrow (q', a)$

of $\mathcal{P}$. The initial graph is constructed from the internal actions and the stack push transitions. New edges are constructed by taking the "transitive closure" of these edges with the stack pop transitions. For example, if $q \overset{a}{\rightsquigarrow} q'$ is an edge in the graph and $(q', a, q'') \in \delta_{\mathrm{rtn}}$ then a new edge $q \overset{\varepsilon}{\rightsquigarrow} q''$ is added to the graph. We modify this algorithm by maintaining the execution time on the labels as well.

▶ **Theorem 13.** *There is a polynomial-time algorithm that, given a PDS* $\mathcal{P} = (Q, \Gamma, \delta)$ *and a set* $Q_0 \subseteq Q$ *of good states, outputs the set* $\{(q, q', m) \mid q \in Q_0$ *and* $(q, \varepsilon) \overset{m}{\Rightarrow}_{\mathcal{P}} (q', \varepsilon)\}$*.*

Proof. The algorithm constructs an edge-labeled directed graph $\mathcal{G}$ iteratively. The set of nodes of $\mathcal{G}$ is $\mathrm{Reach}(Q_0) = \{q \mid \exists q_0 \in Q_0\, \exists w \in \Gamma^* \, (q_0, \varepsilon) \Rightarrow (q, w)\}$. Note that this set can be constructed in polynomial time thanks to Theorem 1. The set of labels on the edges of $\mathcal{G}$ is $\mathbb{N} \times (\Gamma \cup \{\varepsilon\})$. The graph $\mathcal{G}$ is constructed by computing a sequence of graphs $\mathcal{G}_0, \mathcal{G}_1, \ldots$ such that the set of edges of $\mathcal{G}_i$ is a subset of the set of edges of $\mathcal{G}_{i+1}$. The iteration terminates when $\mathcal{G}_i = \mathcal{G}_{i+1}$, in which case $\mathcal{G} = \mathcal{G}_i$. Initially, the set of edges in $\mathcal{G}_0$ is

$$\{(q \xrightarrow{(1,\varepsilon)} q_1) \mid (q, q_1) \in \delta_{\mathrm{int}}\} \cup \{(q \xrightarrow{(1,a)} q_1 \mid (q, q_1, a) \in \delta_{\mathrm{cll}}\}.$$

Assume now that $\mathcal{G}_i$ has been constructed. Then $\mathcal{G}_{i+1}$ is constructed as follows:

- for each pair of edges $q \xrightarrow{(m_1,\varepsilon)} q_1$ and $q_1 \xrightarrow{(m_2,a)} q_2$ in $\mathcal{G}_i$, we add the edge $q \xrightarrow{(m_1+m_2,a)} q_2$;
- for each pair of edges $q \xrightarrow{(m_1,a)} q_1$ and $q_1 \xrightarrow{(m_2,\varepsilon)} q_2$ and in $\mathcal{G}_i$, we add the edge $q \xrightarrow{(m_1+m_2,a)} q_2$;
- for each $a \in \Gamma$, each edge $q \xrightarrow{(m,a)} q_1$ in $\mathcal{G}_i$, and each transition $(q_1, a, q_2) \in \delta_{\mathrm{rtn}}$, we add the edge $q \xrightarrow{(m+1,\varepsilon)} q_2$.

Once $\mathcal{G}$ has been constructed, the algorithm outputs the set

$$\{(q, q, 0) \mid q \in Q_0\} \cup \{(q, q', m) \mid q \in Q_0 \text{ and } q \xrightarrow{(m,\varepsilon)} q' \text{ is an edge of } \mathcal{G}\}.$$

We claim that:

1. The algorithm terminates in polynomial time.
2. The output equals $\{(q, q', m) \mid q \in Q_0 \text{ and } (q, \varepsilon) \overset{m}{\Rightarrow} (q', \varepsilon)\}$. ◀

▶ **Theorem 14.** *The problem of deciding whether the information leaked by the timing behavior of a recursive program P does not exceed q is in* EXPTIME*.*

Proof. As in the case of the proof of Theorem 8, we can assume that P has no low inputs. We can construct the pushdown system corresponding to P and, using Theorem 13, compute the set $R = \{(\bar{h}_0, \mathrm{Steps}_P(\bar{h}_0)) \mid \bar{h}_0 \text{ is a high input}\}$. Now we can partition the set of inputs according to the equivalence relation $\equiv$ defined by $\bar{h}_1 \equiv \bar{h}_2$ iff $\mathrm{Steps}_P(\bar{h}_1) = \mathrm{Steps}_P(\bar{h}_2)$. Let $a_1, \ldots, a_k$ be the partition sizes of $\equiv$. Note that these partition sizes can be computed in time polynomial in the size of the set R, i.e. exponential in the size of P. If m is the number of input variables, then $\sum_1^k a_i = 2^m$ and $\mathrm{SE}_{\mathsf{U}}(\mathrm{Steps}_P) \leq q$ iff $\sigma(a_1, \ldots, a_k) \geq 2^m(m - q)$. (Recall that $\sigma(a_1, \ldots, a_k) = \sum_{i=1}^k a_i \log a_i$.) The latter can now be decided in exponential time as in the proof of Theorem 8. ◀

5 Conclusions and future work

We have considered the problems of checking non-interference and of bounding information leakage in (deterministic) recursive boolean programs with uniformly distributed inputs, proving both problems to be EXPTIME-complete. This implies an EXPTIME upper bound for non-recursive programs, which improves the previously known upper bounds. For the special case when the number of outputs and low inputs is logarithmic in the size of the program, we have established a tight PSPACE upper bound for non-recursive programs.

We have also considered the problem of checking non-interference and of bounding information leakage in recursive boolean programs when the attacker observes the number of execution steps of the program (and not the explicit outputs). Once again, our problems turn out to be EXPTIME-complete in this setting. The proof of the upper bound is interesting from a practical standpoint as we have shown that existing algorithms used for analyzing safety properties in recursive programs can be used for computing information leakage. In fact, we are currently working on a BDD-based symbolic algorithm for computing information leakage in recursive programs.

We have used measures based on Shannon's entropy and mutual information. Nevertheless, our techniques are useful for computing information leakage with respect to other measures. For example, if we use min-entropy to define mutual information [31, 16], the problem of bounding information leakage (from explicit outputs or from timing behavior) for programs with uniformly distributed high inputs is again EXPTIME-complete for recursive programs. We believe that the techniques used in this paper will also be useful for other scenarios, such as the case when we are interested in only the amount of information leaked about certain selected bits of the input.

In addition to extending the results to other scenarios as described above, one particular open problem is to close the gap between the lower bound (PSPACE) and the upper bound (EXPTIME) for non-recursive programs with no restrictions on the number of inputs and outputs. Another interesting direction for future research is to extend our results to programs with probabilistic choices.

Acknowledgements. This research was done while Rohit Chadha was at LSV, ENS Cachan and INRIA Saclay. Michael Ummels was supported by the DFG projects PROBFOR, SYANCO and QuaOS.

References

1 J. Agat. Transforming out timing leaks. In *POPL '00*, pages 40–53, 2000.

2 E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. B. Miltersen. On the complexity of numerical analysis. *SIAM Journal on Computing*, 38(5):1987–2006, 2009.

3 M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

4 G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science (Proc. QAPL '06)*, 153(2):33–55, 2006.

5 A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97*, pages 135–150, 1997.

6 D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

7 P. Černý, K. Chatterjee, and T. A. Henzinger. The complexity of quantitative information flow problems. In *CSF '11*, pages 205–217, 2011.

8 R. Chadha and M. Ummels. The complexity of quantitative information flow in recursive programs. Technical Report LSV-12-15, ENS Cachan, 2012.

9 K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical measurement of information leakage. In *TACAS '10*, pages 390–404, 2010.

10 K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Probability of error in information-hiding protocols. In *CSF '07*, pages 341–354, 2007.

11 K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2-4), 2008.

12 D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science (Proc. QAPL '04)*, 112:49–166, 1984.

13 D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic Computation*, 15(2):181–199, 2005.

14 D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

15 D. E. R. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

16 Y. Dodis, R.Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal of Computing*, 38(1):97–139, 2008.

17 C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL '01*, pages 193–205, 2001.

18 J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

19 J. W. Gray III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.

20 D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):163–182, 2005.

21 A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *Proceedings of the USSR Academy of Sciences*, 145, pages 293–294, 1962.

22 B. Köpf and D. A. Basin. An information-theoretic model for adaptive side-channel attacks. In *ACM Conference on Computer and Communications Security*, pages 286–296, 2007.

23 B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *CSF '09*, pages 324–335, 2009.

24 B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF '10*, pages 3–14, 2010.

25 J. K. Millen. Covert channel capacity. In *IEEE Symposium on Security and Privacy*, pages 60–66, 1987.

26 S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

27 A. D. Pierro, C. Hankin, and H. Wiklicky. Quantifying timing leaks and cost optimisation. In *ICICS '08*, pages 81–96, 2008.

28 T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

29 J. C. Reynolds. Syntactic control of interference. In *POPL '78*, pages 39–46, 1978.

30 C. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423 and 623–656, 1948.

31 G. Smith. On the foundations of quantitative information flow. In *FOSSACS '09*, pages 288–302, 2009.

32 R. van der Meyden and C. Zhang. Algorithmic verification of noninterference properties. *Electronic Notes in Theoretical Computer Science*, 168:61–75, 2007.

33 H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. In *ESORICS '10*, pages 357–372, 2010.

34 H. Yasuoka and T. Terauchi. Quantitative information flow - verification hardness and possibilities. In *CSF '10*, pages 15–27, 2010.

35 H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. In *QAPL 2012*, pages 77–91, 2012.