# Quantitative information flow in Boolean Programs

Rohit Chadha[1], Dileep Kini[2], and Mahesh Viswanathan[2]

[1] University of Missouri
[2] University of Illinois, Urbana-Champaign

**Abstract.** The *quantitative information flow bounding problem* asks, given a program $P$ and threshold $q$, whether the information leaked by $P$ is bounded by $q$. When the amount of information is measured using mutual information, the problem is known to be PSPACE-hard and decidable in EXPTIME. We show that the problem is in fact decidable in PSPACE, thus establishing the exact complexity of the quantitative information flow bounding problem. Thus, the complexity of bounding quantitative information flow in programs has the same complexity as safety verification of programs. We also show that the same bounds apply when comparing information leaked by two programs.

## 1 Introduction

A *non-interferent* program [13, 20] ensures that *low-security observations* by an adversary of an execution of the program is independent of its *high-security* inputs, thus preserving confidentiality of its inputs. While non-interference is desirable, explicit outputs of a program often violate non-interference. For example, the winning bid in an anonymous auction *reveals* information about all other bids, namely an upper bound on other bids. Therefore, others (e.g [12, 14, 19, 22]) propose to quantify the amount of information leaked by a program in order to evaluate security of programs. In these quantitative approaches, a program is seen as a *transformation* of a random variable taking values from the set of inputs into a random variable taking values from the set of observations. The amount of information leaked by the program is modeled as the difference between the initial uncertainty and the uncertainty remaining in the secret inputs given the observations an adversary makes about the execution of the program. In order to measure uncertainty, information-theoretic measures such as Shannon's entropy [12, 14, 19] and min-entropy [22] are employed. The appropriate measure of information usually depends on the application. Min-entropy, for example, is used to measure vulnerability to being guessed in one try and is useful for measuring information leaked by password-checkers. However, note that this is inappropriate for voting protocols which publish vote tallies as an unanimous election always reveals how each voter voted and min-entropy based measure will say that *all* information is leaked for such protocols. Thus, min-entropy will not be able to distinguish a secure electronic protocol from an insecure protocol

protocol such as one which outputs a list of voters along with their voting preferences. Using Shannon entropy to measure information leaked is more appropriate for such protocols.

We consider the complexity of evaluating the amount of information leaked by a program when the uncertainty is measured using Shannon's entropy [21] as has been proposed in [12, 14, 19]. In this case, the amount of information leaked by a program is mutual information between the inputs and outputs. We start by considering the complexity of *quantitative information flow bounding problem* [25, 27]: given a program $P$ with uniformly distributed inputs, and a rational number $q$, check if the information leaked by the program does not exceed $q$.[3] The quantitative information flow bounding problem was first considered in [25, 27] who study complexity of the quantitative information flow problem for (deterministic) imperative Boolean programs. They show that the problem is PP-hard for loop-free Boolean programs. The class PP is the class of decision problems solvable by a probabilistic Turing machine in polynomial time, with an error probability of less than $\frac{1}{2}$. This implies, in particular, that the quantitative information flow bounding problem is harder than reachability in loop-free Boolean programs as the latter is NP-complete and the complexity class PP contains the complexity class NP. Intuitively, the hardness of problem comes from the fact that one has to compute, for each possible output, how many inputs lead to that particular output. For reachability, we only have to guess one input which leads to the reachable state.

For Boolean programs (with loops), the quantitative information flow problem was shown to PSPACE-hard in [25, 27]. However, no upper bounds are given in [25, 27]. The problem was shown to be in EXPTIME in [5]. They also show that the problem is PSPACE-complete when the number of outputs is *logarithmic* in the size of the program.

We briefly recall the strategy used in [5] to establish that the quantitative information flow bounding problem is in PSPACE when the number of outputs is *logarithmic* in the size of the program. The proof therein relies on a recent result on straight-line programs (SLPs). An SLP is a sequence of assignments to integer variables in which the operations allowed are addition, subtraction and multiplication (no division). The value of the variable last assigned to is said to be the number defined by SLP. A recent result shows that the problem of checking whether an SLP defines a strictly positive number is decidable in counting hierarchy [2], which is contained in PSPACE. Now, [5] show that for each program $P$ and rational number $q$ there is an SLP $prog_{(P,q)}$ such that the information leaked by the program by program $P$ does not exceed $q$ iff the number defined by $prog_{(P,q)}$ is strictly positive. The program $prog_{(P,q)}$ is polynomial in a size of $P$ and $q$ if the number of outputs of $P$ is logarithmic in the size of the program (and can be constructed in polynomial time). However, if there is no restriction on number of outputs then the size of $prog_{(P,q)}$ can

[3] Uniformly distributed inputs is a commonly used assumption. For arbitrary input distributions, the same complexity bounds will apply for Boolean programs.

be exponential in the size of $P$. Therefore, they have to restrict the number of outputs to achieve the PSPACE upper bound.

**Contributions.** Our first contribution is to show that the quantitative information flow bounding problem is in PSPACE which matches the PSPACE lower bound without any restrictions on the number of outputs, thus establishing the *exact* complexity of quantitative information flow. This is surprising since checking safety of Boolean programs (or reachability in Boolean programs) is also PSPACE-complete. This shows that the problem of bounding quantitative information flow is (complexity-theoretically) as easy as safety verification of Boolean programs.

For the upper bound, we cannot directly use the construction of the SLP $prog_{(P,q)}$ outlined in [5] as the size of $prog_{(P,q)}$ is exponential in the size of $P$. Instead, we establish a new result about PSPACE-SLP generators. A PSPACE-SLP generator $F$ is an algorithm that outputs an SLP on its input $w$, using only a polynomially-bounded work-tape (note the output can still be exponential in $|w|$, the length of $w$). We give sufficient conditions that ensure that the problem of checking whether, given $w$, the number defined by the SLP $F(w)$ is strictly positive can be decided in PSPACE (even when the output $F(w)$ is exponential in length of $w$). We then show that there is a PSPACE-SLP generator $f$ that a) satisfies the above conditions and b) given a program $P$ and a rational number $q$ computes the SLP $prog_{(P,q)}$.

We then consider the *quantitative information flow comparison problem*: given programs $P_1$ and $P_2$ check if information leaked by program $P_1$ is less than the information leaked by $P_2$. The *quantitative information flow comparison problem* was first studied in [26], where they show that the comparison problem is #P-hard for loop-free Boolean programs.[4] We show that the *quantitative information flow comparison problem* is also PSPACE-complete by using methods similar to the quantitative information flow bounding problem.

Finally, we are able to conclude PSPACE-completeness for quantitative information flow bounding problem and the quantitative information flow comparison problem when the observations of the adversary are not explicit outputs, but implicitly derived from the timing behavior of an execution of a program. Following [5], we abstract the timing behavior as the number of steps of the execution of a program. The conclusion follows from the observation that a Boolean program takes at most an exponential number of steps[5] and hence can be encoded by a binary counter.

The rest of the paper is organized as follows. We introduce relevant notation and definitions in Section 2. We establish our result on SLP-generators in Section 4. The results on quantitative information flow bounding problem and

[4] Note #P is a class of function problems and not decision problems. The precise statement of #P-hardness in [26] is that the function class #P is as hard as the class of function problems solvable in polynomial time with an oracle for comparing quantitative information flow in loop-free Boolean programs.

[5] We only consider terminating programs.

quantitative information flow comparison problem are presented in Section 5. We conclude in Section 6.

**Related work.** In the recent years, several automated approaches from model-checking [3, 18, 7, 8], static analysis [9–11, 3] and statistical analysis [18, 6] have been employed to compute the information leaked by a program. The *complexity* of computing the amount of leakage was only considered recently [26, 25, 27, 24, 5]. The problem was first tackled in [26], where quantitative information flow comparison problem is studied. The PSPACE lower bound for non-interference was shown in [27] which implies the lower bound for both the quantitative information flow bounding problem and for the quantitative information flow comparison problem. A PSPACE upper and lower bound for programs for measures based on min-entropy and guessing entropy was established in [27]. However, an exact upper bound was not known for the case when the information is measured using Shannon's entropy.

Non-interference and quantitative information flow bounding problem when programs are modeled abstractly as nondeterministic transition systems are considered in [23] and [24] respectively. In this setting, the problems are shown to be PSPACE-complete.[6] However, this only implies an EXPSPACE-upper bound because the translation into an explicit state description causes an exponential blowup. For example, the following program with three variables $x, y, z$:

$$x := x \vee (y \wedge z);\ \text{end}$$

is represented in [24] as a state machine with 16 states. (The factor of 2 is due to the program counter).

As discussed above, the best known upper bound for quantitative information flow bounding problem was EXPTIME [5] which follows from EXPTIME-completeness of quantitative information flow problem in recursive Boolean programs [5]. The EXPTIME upper bound improves to PSPACE when the number of outputs variables are logarithmic in the size of the program [5].

The proof of the PSPACE-upper bound in [5] is established by reducing the quantitative information flow bounding problem in PSPACE to checking whether an SLP defines a positive integer. The restriction on the number of outputs ensures that the constructed program is polynomial in length. Then the recent result of [2] is invoked which shows that the problem of checking whether an SLP defines a positive integer is in counting hierarchy and hence in PSPACE. Since we are not restricting the number of outputs, the reduction yields an SLP which is exponential in input size. Thus, we cannot use [2] and have to establish our result on SLP generators.

Attacks based on implicit observations of program execution, such as timing behavior, are hard to protect against and can lead to serious breaches of confidentiality (see for example, [4] which exhibits a practical timing attack against

[6] Even though the partial program model defined in [24] is a Boolean program, the complexity results are reported in terms of the size of the explicit nondeterministic transition system, which is exponentially larger than the size of partial program.

OpenSSL which allows the adversary to obtain private RSA keys.) Hence, several approaches have been proposed in the literature to counteract timing leaks (see [1, 17] for example).

## 2 Preliminaries

We recall some standard definitions and establish some notations. Note that for a set $X$, the set of all boolean valued functions with domain $X$ shall be denoted as $2^X$. Please note that our notations closely follow [5].

### 2.1 Boolean Programs

*Syntax:* The programs that we consider have input variables, output variables and local variables. Input variables can be either *high-security input variables*, meaning that the adversary cannot observe their values, or *low-security input variables*, meaning that their values are known to the adversary. The output variables will be *low-security variables*, i.e., their values shall be observable to the adversary. The values of local variables will not be observable to the adversary.[7]

Formally, we assume a countable set Vars of variables which can take Boolean values $\top$ (true) and $\bot$ (false). The set Exps of Boolean expressions is generated by the following BNF grammar ($x \in \text{Vars}$):

$$\phi ::= \top \mid \bot \mid x \mid \neg\phi \mid (\phi \vee \phi) \mid (\phi \wedge \phi)\,.$$

A program can manipulate its variables using statements. The set of statements, Stmts, is defined by the following BN F grammar ($x \in \text{Vars}$, $\phi \in \text{Exps}$):

$$\begin{array}{rll} s ::= & \textbf{skip} & \text{(Skip)} \\ & \mid x \leftarrow \phi & \text{(Assignment)} \\ & \mid \textbf{if } \phi \textbf{ then } s \textbf{ else } s \textbf{ end} & \text{(Conditional)} \\ & \mid \textbf{while } \phi \textbf{ do } s \textbf{ end} & \text{(Iteration)} \\ & \mid s; s & \text{(Sequential composition)} \end{array}$$

As usual we say that $\text{Vars}(s)$ is the set of variables occurring in $s$.

A *program* $P$ is of the form

$$\textbf{high } \overrightarrow{h};\ \textbf{low } \overrightarrow{l};\ \textbf{out } \overrightarrow{o};\ \textbf{local } \overrightarrow{z};\ s$$

where $s$ is a statement and $\overrightarrow{h}, \overrightarrow{l}, \overrightarrow{o}, \overrightarrow{z}$ are vectors of variables such that $\text{Vars}(s) \subseteq \overrightarrow{h} \cup \overrightarrow{l} \cup \overrightarrow{o} \cup \overrightarrow{z}$.

[7] Note that high-security output variables can always be modeled as local variables.

*Semantics:* Recall that a transition system, $T$, is a tuple $(Q, \rightarrow)$ where the set $Q$ is a finite set of *configurations* and $\rightarrow \subseteq Q \times Q$ is a set of transitions. $T$ is said to be *deterministic* if $c \rightarrow c_1$ and $c \rightarrow c_2$ implies that $c_1 = c_2$. A *computation* from a configuration $c_0$ is a sequence $c_0 \rightarrow \cdots \xrightarrow{c}_m$. We say that $c \overset{m}{\Rightarrow} c'$ if there exists a computation $c_0 \rightarrow \cdots \rightarrow c_m$ with $c_0 = c$ and $c_m = c'$, and we write $c \Rightarrow c'$ if $c \overset{m}{\Rightarrow} c'$ for some $m \in \mathbb{N}$.

We give an informal description of the semantics of programs. The operational semantics of a program $P$ can be given in terms of a deterministic transition system $(\mathrm{Conf}_P, \rightarrow_P, c_0)$ of size exponential in the size of $P$. A configuration $c \in \mathrm{Conf}_P$ keeps track of the "current line number" and the "values" of the variables of the program $P$. A transition in $\rightarrow_P$ represents one execution step of $P$. The program $P$ *terminates* on inputs $\overrightarrow{h}_0, \overrightarrow{l}_0$ if there is a computation from a configuration in which the "current line number" is the line of the first statement of $P$, the input variables are set to $\overrightarrow{h}_0, \overrightarrow{l}_0$, and the local and output variables are set to $\perp$ that reaches the configuration with the "current line number" corresponding to line number of the last statement of the program. If $P$ terminates, we define the output of $P$ to be the values of the output variables upon termination.

Therefore, $P$ can be seen as a partial function $F_P \colon H \times L \rightarrow O$ where $H = 2^{\overrightarrow{h}}, L = 2^{\overrightarrow{l}}$ and $O = 2^{\overrightarrow{o}}$. $F_P(\overrightarrow{h_0}, \overrightarrow{l_0})$ is defined iff $P$ terminates on $\overrightarrow{h_0}, \overrightarrow{l_0}$, and is the value output by $P$ on $\overrightarrow{h_0}, \overrightarrow{l_0}$. From now on, we will confuse $P$ with the function $F_P$. We will only consider terminating programs. 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.

## 3 Mutual Information

We recall some standard definitions. Let $\mathcal{X}$ be a discrete random variable with values taken from a finite set $X$. If $\mu$ is the probability distribution of $\mathcal{X}$, the *Shannon entropy* of $\mu$, 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 $\mu$, the *conditional entropy* of $\mathcal{X}$ given $\mathcal{Y}$, written $\mathrm{H}_\mu(\mathcal{X} \mid \mathcal{Y})$, is defined as

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

The *mutual information* of $\mathcal{X}$ and $\mathcal{Y}$, written $\mathrm{I}_\mu(\mathcal{X}; \mathcal{Y})$, is defined as

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

We have $\mathrm{I}_\mu(\mathcal{X}; \mathcal{Y}) \geq 0$.

## 3.1 quantitative information flow in programs

We use conditional mutual information to quantify the amount of information leaked by the program as has been proposed in [12, 14, 19]. We assume that the reader is familiar with information theory and in particular conditional mutual information. As discussed above, the semantics of a Boolean program $P$ is a function $P : H \times L \to O$. Assume now that the inputs are sampled from a distribution $\mu$. Let $\mathcal{H}$ be the random variable taking values in $H$ and $\mathcal{L}$ be the random variable taking values in $L$ according to the distribution $\mu$. $\mu$ can be extended to a joint probability distribution on $H$, $L$ and $O$ as follows

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

The *information leaked by the program* $P$ is then defined to be

$$\mathrm{SE}_\mu(P) \coloneqq \mathrm{I}_\mu(\mathcal{H}; \mathcal{O} \mid \mathcal{L}).$$

In case there are no low-security inputs, the information leaked by the function $F$ is just the mutual information between $\mathcal{H}$ and $\mathcal{O}$:

$$\mathrm{SE}_\mu(P) = \mathrm{I}_\mu(\mathcal{H}; \mathcal{O}).$$

A program $P$ is *non-interferent* iff $\mathrm{SE}_\mu(P) = 0$ for all $\mu$.[8]

We now recall a result proved in [5] that will allow us to restrict our attention to programs that have only high-security inputs. Given a program $P$ with high-security input variables $\overrightarrow{h}$, low-security input variables $\overrightarrow{l}$ and low-security output variables $\overrightarrow{o}$, let the program $P_{nolowinp}$ be defined as follows. For each variable $l \in \overrightarrow{l}$, pick a new variable $l_{new}$. $P_{nolowinp}$ has high-security input variables $\overrightarrow{h}, \overrightarrow{l}$ and no low-security input variables. The output variables of $P_{nolowinp}$ are $\overrightarrow{o}$ and $\overrightarrow{l_{new}}$. The program $P_{nolowinp}$ initially copies the values $\overrightarrow{l}$ into $\overrightarrow{l_{new}}$ and then behaves exactly like $P$. The following is shown in [5]:

**Proposition 1.** $\mathrm{SE}_\mu(P_{nolowinp}) = \mathrm{SE}_\mu(P) + \mathrm{H}_\mu(\mathcal{L})$.

Note that when $\mu$ is $\mathsf{U}$, the uniform distribution on $H \times L$, we have that $\mathrm{H}_\mu(\mathcal{L}) = \log |\mathcal{L}|$. Thus, it follows that if $P$ has $r$ low-security input variables, we get that

**Proposition 2.** $\mathrm{SE}_\mathsf{U}(P_{nolowinp}) = \mathrm{SE}_\mathsf{U}(P) + r$.

Thus, for uniformly distributed inputs, we shall only need to consider programs with no low-security inputs. We shall make use of the following theorem proved in [3, 16]:

**Theorem 1.** *Let $F_P \colon H \to O$ be the semantics of a program $P$ with no low-security inputs. Then*

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

[8] It can be shown that this definition is equivalent to the standard definition of non-interference: for any low-input $l \in L$ and high inputs $h, h' \in H$, we have that $P(h,l) = P(h',l)$.

### 3.2 Decision problems for quantitative information flow

*The quantitative information flow bounding problem:* The *quantitative information flow bounding problem* asks, given a program $P$ and a rational number $q \geq 0$, whether the information leaked by $P$ does not exceed $q$, i.e., whether $\mathrm{SE}_{\mathsf{U}}(P) \leq q$. The input to the decision problem is the program $P$ and $q$ (with numerator and denominator given in binary). The size of input problem is the size of $P$ and the size of numerator and denominator $q$.

*The quantitative information flow comparison problem:* The *quantitative information flow comparison problem* asks, given programs $P$ and $P'$, whether the information leaked by $P$ exceeds the information leaked by $P'$, i.e., whether $\mathrm{SE}_{\mathsf{U}}(P') < \mathrm{SE}_{\mathsf{U}}(P)$. The input to the decision problem are the programs $P$ and $P'$.

### 3.3 Straight-line programs (SLP)s.

Let $Var$ be a countable set of variables. A (division-free) *straight-line program (SLP)* is a finite sequence of statements of the form $x \leftarrow 0$ or $x \leftarrow 1$ or $x \leftarrow Y \odot Z$, where $\odot \in \{+, -, \cdot\}$, $x \in Var$ and $Y, Z \in Var \cup \{0, 1\}$. are taken from a countable set of variables.

An SLP $p$ is said to be *closed* if each variable that appears on the right-hand side of a statement also appears on the left-hand side of a preceding statement. The semantics for any such program is the usual where $\leftarrow$ corresponds to assignment and the operators $+,-,\cdot$ are addition, subtraction and multiplication over the set of integers, $\mathbb{Z}$. The value of a closed $\mathsf{SLP}$ $p$ denoted by $val(p)$ is the value of the last variable assigned in its last statement. The problem PosSLP is to decide, given a closed SLP $p$, whether $val(p) > 0$. It is shown in [2] that PosSLP is in counting hierarchy.

The standard square-and-multiply algorithm for exponentiation gives us the following.

**Proposition 3.** *Given a natural number $N > 0$ in binary with $k$ bits, there is a closed SLP $p_N$ of length $O(k)$ using only one variable such that $val(p_N) = N$. Given numbers $N_1, N_2 > 0$ in binary with $k_1$ and $k_2$ bits respectively, there is a closed SLP $p_{N_1^{N_2}}$ of length $O(k_1 + k_2)$ and using only 2 variables such that $val(p_{N_1^{N_2}}) = N_1^{N_2}$.*

## 4 SLP generators

Given a finite set of variables $V$, we will say that a *SLP generator compatible with $V$* is an algorithm that outputs a closed SLP which uses variables in $V$. We shall be interested in special kinds of SLP generators:

**Definition 1.** *An SLP generator $f$ compatible with $V$ is said to be a good $\mathsf{SLP}$-generator if:*

- *$f$ is a function computable in* PSPACE*, i.e., for any input $w$, $f(w)$ is computed using polynomially bounded workspace.*
- *For any input $w$, if the SLP $f(w)$ were to be executed then for each $x \in V$ at any point, the value of the variable $x$ is $\geq -2^{2^{|w|}}$ and is $< 2^{2^{|w|}}$, where $|w|$ is the length of $w$.*

Please note that the output $f(w)$ generated using a good SLP-generator $f$ can be exponentially long (in terms of $|w|$), as will be the case when we apply our results in Theorem 2. We will be interested in deciding whether for a given $w$, the value of the program $f(w) > 0$. Since, $f(w)$ is exponentially long, we cannot directly apply result of [2] which says that PosSLP is in counting hierarchy and hence in PSPACE. However, the conditions of being a good SLP generator will allow us to adapt the proof of PosSLP being in counting hierarchy to establish the following result, whose proof has been moved to the Appendix for the sake of the flow of the paper.

**Lemma 1.** *Given a good* SLP *generator $f$ compatible with $V$, the following language is* PSPACE*:*

$$\{w \mid val(f(w)) > 0\}.$$

## 5 Complexity of quantitative information flow

We will now consider the quantitative information flow bounding problem and the quantitative information flow comparison problem, showing both of them to be PSPACE-complete. Since non-interference in Boolean programs is PSPACE-hard [27], these two problems are easily seen to be PSPACE-hard. We only need to show the upper bounds.

We start by first considering the quantitative information flow bounding problem. We shall need one Lemma.

**Lemma 2.** *Let $P$ be a program with $\overrightarrow{h}$ as high-security input variables, no low-security input variables and $\overrightarrow{o}$ as the output variables. If $n$ is the number of high-security input variables and $O = 2^{\overrightarrow{o}}$ then*

$$\prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|} \leq 2^{n \cdot 2^n}.$$

*Proof.* Since mutual information is always positive, we have that $\mathrm{SE}_{\mathsf{U}}(P) \geq 0$. Thus, by Theorem 1, we get that

$$n - \frac{1}{2^n} \log \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|} \geq 0.$$

Therefore,

$$\log \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|} \leq n2^n.$$

The claim follows by exponentiating both sides. □

**Theorem 2.** *The quantitative information flow bounding problem is* PSPACE-*complete for Boolean programs.*

*Proof.* We only need to prove the upper bound. Thanks to Proposition 2, we only need to consider programs with no low-security inputs.

Let $P$ be a program with high input variables $\overrightarrow{h}$ and low-security output variables $\overrightarrow{o}$. Let the number of input variables be $n$ and the number of output variables be $m$. Let $H = 2^{\overrightarrow{h}}$ and $O = 2^{\overrightarrow{o}}$. Let $q$ be a rational number. We have $|H| = 2^n$.

Theorem 1 implies that $\mathrm{SE}_{\mathsf{U}}(P)$, the information leaked by $P$, is

$$n - \frac{1}{2^n} \log \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|}.$$

Thus,

$$\mathrm{SE}_{\mathsf{U}}(P) \leq q \Leftrightarrow \log \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|} \geq 2^n(n-q).$$

Observe that the number $2^n(n-q)$ is polynomial in the size of the program $P$ and rational number $q$ and can be computed in polynomial time. Thus it suffices to show that for any given positive rational number $\frac{r}{s}$, we can decide if

$$\log \prod_{o \in O} |P^{-1}(o)| \log |P^{-1}(o)| \geq \frac{r}{s}$$

in polynomial space ($r$ can be taken to be positive as a program never leaks more than $n$ bits). Note that since

$$\log \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|} \geq \frac{r}{s} \Leftrightarrow \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)| \cdot s} \geq 2^r,$$

it suffices to show that we can decide if

$$\prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)| \cdot s} - 2^r \geq 0$$

in polynomial space.

In order to show this, we will construct a good $\mathsf{SLP}$ generator $f$ that given a program $P$ and natural numbers $r$ and $s$, constructs a $\mathsf{SLP}$ program with 6 variables[9] $S$ such that

$$val(S) = \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)| \cdot s} - 2^r + 1.$$

The result will then follow from Lemma 1.

Let $k$ be the size of $s$. We make a few observations before we show how to construct $f$.

[9] Since there are exponentially many outputs, program $S$ itself will turn out to be exponential in length.

(a) $|P^{-1}(o)| \leq 2^n$. Hence $|P^{-1}(o)|$ can be represented as a binary number of size $n+1$. The number $|P^{-1}(o)|s$ can be represented as a number of size $n+k+1$ and computed in polynomial time given $|P^{-1}(o)|$ and $s$.

(b) Given $|P^{-1}(o)|$ and $|P^{-1}(o)|s$, we can construct in polynomial time (and hence in polynomial space) a $\mathsf{SLP}$ $S_o$ whose value is $|P^{-1}(o)|^{|P^{-1}(o)|\cdot s}$ using two variables (See Proposition 3).

(c) By Lemma 2,

$$\prod_{o\in O} |P^{-1}(o)|^{|P^{-1}(o)|\cdot s} \leq 2^{s\cdot n\cdot 2^n}$$

and hence

$$\prod_{o\in O} |P^{-1}(o)|^{|P^{-1}(o)|\cdot s} \leq 2^{n\cdot 2^{n+k}}.$$

(d) $P$ has $m$ output variables. Let us fix an enumeration $o_1, \ldots, o_m$ of these variables. Hence each possible output $o \in O$ can be uniquely identified with a $m$-bit binary natural number whose $\ell$-th bit is 1 iff $o_\ell$ is 1. We will henceforth confuse elements of $O$ with the $m$-bit binary numbers representing them. We will use $j$ to range over the $m$-bit binary natural numbers representing elements of $O$. Similarly, each input of $P$ can be identified with a $n$-bit natural number.

(e) For any output $j$, $|P^{-1}(j)|$ can be computed using a work-tape polynomial in the size of $P$ as follows. We initialize $|P^{-1}(j)|$ as 0. Recall that every input can be represented as a $n$-bit integer. We utilize this to iterate over all inputs as follows. We initialize a $n$-bit integer $k$ as 0. At the 1-st iteration, we take all inputs to be zero, run the program $P$ on this (which can be done in polynomial space) and check whether the output of $P$ is $j$ or not. If the output is $j$ then we increment $|P^{-1}(j)|$ otherwise we leave $|P^{-1}(j)|$ unchanged. In either case, we increment $k$. At $k+1$-st iteration, we take the input corresponding to $k$, run the program $P$ on this input and check whether the output of $P$ is $j$ or not. Once again, if the output is $j$ we increment $|P^{-1}(j)|$ otherwise we leave $|P^{-1}(j)|$. In either case, we increment $k$. The iterations stop when $k$ becomes $2^n$.

Now, we describe how the good $\mathsf{SLP}$ generator $f$ is constructed. $f$ will use six variables $\{z, x_0, y_0, x_1, y_1, res\}$ and its input tape will have a definition of $P$ along with natural numbers $r$ and $s$ written on it. We give the psuedo-code for $f$ in Figure 1 and describe $f$ in detail. The PSLP $f$ uses two integers $j$ and $k$. $j$ is used to iterate over outputs. In each iteration, the integer *count* is used to compute the number of inputs that lead to the output $j$. The computation of *count* is done by iterating over all inputs ($k$ is used for this iteration).

At the first step $f$ will output the assignment $z \leftarrow 1$. Now, $f$ will do $2^m$ iterations numbered $0, 1, \ldots, 2^{m-1}$. At iteration $j$, $f$ first computes $|P^{-1}(j)|$. Note that as observed above in (e), $|P^{-1}(j)|$ can be computed using polynomial workspace by iterating over all possible inputs, running program $P$ for each input. After computing $|P^{-1}(j)|$, $f$ computes $|P^{-1}(j)|s$ and $f$ outputs the SLP computing $|P^{-1}(j)|^{|P^{-1}(j)|\cdot s}$ using variables $x_0$ and $y_0$. Without loss of generality,

```
Input: $P, r, s$ where
          $P$ is a program, $r$ and $s$ are natural numbers.
          Let $h_0, h_1, \cdots, h_n$ be the input variables of $P$ and
           $o_0, o_1, \cdots, o_m$ be the output variables of $P$.
{
int k,j,count,power;
Output(“$z \leftarrow 1$”);
for ($j := 0, j < 2^m, j++$)
  {
      $count := 0$;
      for ($k := 0, k < 2^n, k++$)
          {
            $h_0 := k[0]; h_1 := k[1]; \cdots; h_n := k[n]$;
            $P(\overrightarrow{h})$;
            if $(o_0 = j[0]) \wedge (o_1 = j[1]) \wedge \cdots \wedge (o_m = j[m])$
            then $count := count + 1$
          }
      $power := count \times s$;
      Output($expSLP(count, power, x_0, y_0)$);
      Output(“$z \leftarrow z \cdot y_0$”);
  }
Output($expSLP(2, r, x_1, y_1)$);
Output(“$y_1 \leftarrow y_1 - 1$”);
Output(“$res \leftarrow z - y_1$”);
}
```

**Fig. 1.** Psuedo-code for the good SLP generator $f$. The $\ell$-th bit of $j$ ($k$, respectively) is represented by $j[\ell]$ ($k[\ell]$, respectively). The command Output($str$) outputs the string $str$. $expSLP(t, u, x, y)$ is the SLP computing $t^u$ using variables $x, y$ with result being stored in $y$ (See Proposition 3).

we can assume that the variable assigned to in the last statement of the program for $|P^{-1}(j)|^{|P^{-1}(j)|s}$ is $y_0$. After outputting the SLP defining $|P^{-1}(j)|^{|P^{-1}(j)|s}$, $f$ outputs the assignment $z \leftarrow z \cdot y_0$. Using observations (a) and (b) above, it is easy to see all these $2^m$ iterations can be done using polynomially bounded space.

After the $2^m$ iterations are over, it is easy to see the SLP output thus far has value $\prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)| \cdot s}$ and that the variable last assigned to is $z$. $f$ next outputs the SLP for $2^r$ using variables $x_1$ and $y_1$ with $y_1$ being the variable assigned to in the last statement of the SLP for $2^r$. $f$ then outputs the assignment $y_1 \leftarrow y_1 - 1$.

Next, $f$ outputs the statement $res = z - y_1$ and then terminates. It is easy to see using observation $(c)$ above that $f$ is a good SLP generator. and that if $S$ is the SLP output by $f$ then $val(S) = \prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)| \cdot s} - 2^r + 1$. The result follows from Lemma 1. □

Similarly we can show that the quantitative information flow comparison problem is PSPACE-complete.

**Corollary 1.** *The quantitative information flow comparison problem is PSPACE-complete for Boolean programs.*

*Proof.* First, we make an observation that we will allow us to assume that the programs being compared have the same number of input variables. If $P$ is a program with $n$ inputs, let $P_{moreinputs}$ be the program with $n+k$ inputs which is constructed from $P$ as follows. $P_{moreinputs}$ has exactly the same set of low-security input variables and the same set of output variables as $P$. $P_{moreinputs}$ also has each high-security input variable that has $P$ has. In addition $P$ has $k$ new high security inputs $o^1_{new}, o^2_{new}, \ldots, o^k_{new}$. The body of the program $P_{moreinputs}$ is exactly the body of the program $P$ (in other words, $o^1_{new}, o^2_{new}, \ldots, o^k_{new}$ are never utilized in the program). Then using Theorem 1 and Proposition 2, it follows that $\mathrm{SE}_{\mathsf{U}}(P) = \mathrm{SE}_{\mathsf{U}}(P_{moreinputs})$.

As observed above, we only need to show the upper bound. Now, let $P$ and $P'$ be two programs with the same number of input variables $m$ and with $O$ and $O'$ as the set of outputs respectively. If $r_1$ and $r_2$ are the number of low-security input variables of $P_1$ and $P_2$ respectively then it is easy to see using Theorem 1 and Proposition 2 that

$$\mathrm{SE}_{\mathsf{U}}(P) < \mathrm{SE}_{\mathsf{U}}(P')$$

iff

$$\begin{aligned} & m - \tfrac{1}{2^m} \log(\textstyle\prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|}) - r_1 < \\ & \qquad m - \tfrac{1}{2^m} \log(\textstyle\prod_{o' \in O'} |P'^{-1}(o')|^{|P'^{-1}(o')|}) - r_2 \end{aligned}$$

iff

$$\begin{aligned} & \log(\textstyle\prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|}) > \\ & \qquad \log(\textstyle\prod_{o' \in O'} |P'^{-1}(o')|^{|P'^{-1}(o')|}) + (r_2 - r_1)2^m \end{aligned}$$

iff

$$\prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|} > 2^{(r_2 - r_1)2^m} \prod_{o' \in O'} |P'^{-1}(o')|^{|P'^{-1}(o')|}.$$

Note that $2^m$ is representable by a $m$-bit integer. The result follows by observing that we can check if

$$\prod_{o \in O} |P^{-1}(o)|^{|P^{-1}(o)|} > 2^{(r_2 - r_1)2^m} \prod_{o' \in O'} |P'^{-1}(o')|^{|P'^{-1}(o')|}.$$

in PSPACE in a fashion similar to the proof of Theorem 2. □

### 5.1 Information leaked from timing behavior

We now turn our attention to the problem of estimating the information leaked by a program by its timing behavior. [5] propose using the number of steps taken by a program as an abstraction of timing behavior. Thus, in order to measure the information leaked by a program by its timing behavior, we can consider $P$ as function from its inputs to natural numbers and define the amount of information leaked by the timing behavior as in Section 3.1. More precisely, given a program $P$, with high input variables $\overrightarrow{h}$ and low input variables $\overrightarrow{l}$, let $\mathrm{Steps}_P : 2^{\overrightarrow{h}} \times 2^{\overrightarrow{l}} \to \mathbb{N}$ be the function such that $\mathrm{Steps}_P(\overrightarrow{h}_0, \overrightarrow{l}_0)$ is the number of steps in the computation of $P(\overrightarrow{h}_0, \overrightarrow{l}_0)$. Now, we can define $\mathrm{SE}_\mu(\mathrm{Steps}_P)$ in a manner analogous to the definition of $\mathrm{SE}_\mu(P)$.

**Definition 2.** $\mathrm{SE}_{\mathsf{U}}(\mathrm{Steps}_P)$ *is the* information leaked by the timing behavior *of $P$.*

A terminating while program takes at most $c2^t$ steps, where $c$ is the number of statements in the program and $t$ is the total number of variables of the program. The number of steps can be represented as a natural number whose (binary) size is polynomial in the size of program. It is easy to see that the proofs of Theorem 2 and Corollary 1 can be modified to show the following (the lower bounds follow from [5]):

**Corollary 2.** *The problem of bounding information leaked by the timing behavior of a Boolean program is PSPACE-complete. The problem of comparing information leaked by timing behavior of Boolean programs is PSPACE-complete.*

## 6 Conclusions and Future work

We have shown that the quantitative information flow bounding problem and the quantitative information flow comparison problem for Boolean programs are PSPACE-complete. Surprisingly, this matches the PSPACE-completeness of safety verification of Boolean programs. The same bounds apply when the adversary observes the number of executions steps and not explicit outputs.

The PSPACE-upper bound result implies that the quantitative information flow bounding problem is reducible to safety verification of Boolean programs. While we have not given this direct reduction, one can be obtained by composing the reductions in the proofs of Lemma 1 and Theorem 2. Thus, in order to check if the information leaked by a program is less than $q$ we can reduce the problem to the safety verification problem of Boolean programs and then use an off-the-shelf verification tool. We are currently investigating this approach.

In order to establish the upper bound, we establish a new result on SLP generators. In particular, we give sufficient restrictions that ensures that the problem of checking whether the number defined by the output of an SLP generator $F$ on input $w$ is strictly positive can be decided in PSPACE. This result may be of independent interest.

*Acknowledgements.* Work by Rohit Chadha was supported by the National Science Foundation under Grant No. TWC-1314338. Work by Dileep Kini was supported by the National Science Foundation under Grant No. SHF-1016989. Work by Mahesh Viswanathan was supported by the National Science Foundation under Grant No. TWC-1314485.

## References

1. Johan Agat. Transforming out timing leaks. In *POPL '00*, pages 40–53, 2000.
2. Eric Allender, Peter Bürgisser, Johan Kjeldgaard-Pedersen, and Peter Bro Miltersen. On the complexity of numerical analysis. *SIAM Journal on Computing*, 38(5):1987–2006, 2009.
3. Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*, pages 141–153, 2009.
4. David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
5. R. Chadha and M. Ummels. The complexity of quantitative information flow in recursive programs. In *FSTTCS*, pages 534–545, 2012.
6. Konstantinos Chatzikokolakis, Tom Chothia, and Apratim Guha. Statistical measurement of information leakage. In *TACAS '10*, pages 390–404, 2010.
7. Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Probability of error in information-hiding protocols. In *CSF '07*, pages 341–354, 2007.
8. Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2-4), 2008.
9. David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science (Proc. QAPL '04)*, 112:49–166, 1984.
10. David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic Computation*, 15(2):181–199, 2005.

11. David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.
12. Dorothy E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
13. Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
14. James W. Gray III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.
15. W. Hesse, E. Allender, and D. A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computing Systems and Sciences*, 65(4):695–716, 2002.
16. Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In *ACM Conference on Computer and Communications Security*, pages 286–296, 2007.
17. Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *CSF '09*, pages 324–335, 2009.
18. Boris Köpf and Andrey Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF '10*, pages 3–14, 2010.
19. Jonathan K. Millen. Covert channel capacity. In *IEEE Symposium on Security and Privacy*, pages 60–66, 1987.
20. John C. Reynolds. Syntactic control of interference. In *POPL '78*, pages 39–46, 1978.
21. Claude Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423 and 623–656, 1948.
22. Geoffrey Smith. On the foundations of quantitative information flow. In *FOSSACS '09*, pages 288–302, 2009.
23. Ron van der Meyden and Chenyi Zhang. Algorithmic verification of noninterference properties. *Electronic Notes in Theoretical Computer Science*, 168:61–75, 2007.
24. Pavol Černý, Krishnendu Chatterjee, and Thomas A. Henzinger. The complexity of quantitative information flow problems. In *CSF '11*, pages 205–217, 2011.
25. Hirotoshi Yasuoka and Tachio Terauchi. On bounding problems of quantitative information flow. In *ESORICS '10*, pages 357–372, 2010.
26. Hirotoshi Yasuoka and Tachio Terauchi. Quantitative information flow - verification hardness and possibilities. In *CSF '10*, pages 15–27, 2010.
27. Hirotoshi Yasuoka and Tachio Terauchi. Quantitative information flow as safety and liveness hyperproperties. In *QAPL 2012*, pages 77–91, 2012.

## A Proof of Lemma 1

We recall a couple of definitions before we give the proof of the lemma.

*Chinese Remainders:* First we recall the the idea of Chinese Remainder Representation, CRR, of a number: An $m$-bit number can be uniquely represented using its residue modulo polynomially many primes each of which is $O(\log m)$ bits.[10]. More precisely, if we are given the residues of a number modulo all primes $p < m^2$ then there is a unique such number among those with less than $m$-bits. By the $(p, j)$-th bit of a CRR of a number $N$, we will mean the $j^{\text{th}}$ bit of the residue with respect to prime $p$.

[10] A residue of a number $N$ modulo a prime $p$ is the remainder when $N$ is divided by $p$

*Dlogtime-uniform threshold circuits:* A *majority gate* is one which outputs the value which occurs in most of its inputs. Boolean Circuits built using majority gates in addition to the boolean gates are called *threshold* circuits. The uniformity condition of Dlogtime essentially says that the connection between gates of the circuit can be determined in logarithmic time.

Let $f$ be a good SLP-generator and let $w$ be an input word of length $n$. Now, for SLP $f(w)$, we know that the integers to which the variables evaluate to are always between $-2^{2^n}$ to $2^{2^n}-1$. So if we define an SLP $X_w$ which computes $2^{2^n}+val(f(w))$ then $val(X_w) \geq 0$. Furthermore, $val(f(w)) > 0$ iff the most significant bit of $val(X_w)$ and some other bit of $val(X_w)$ is 1. Also note that $X_w$ uses 2 extra variables (first compute $2^{2^n}$ using two new variables, then follow it by the SLP $f(w)$, and finally add the result of $2^{2^n}$ to $f(w)$). We can of course fix the two extra variables (i.e., the same extra two variables will work for all $w$).

Now we are ready to state the crucial result from [15] which we shall use in the proof:

**Theorem 3.** *There are Dlogtime-uniform threshold circuits $D_m$ of polynomial size and constant depth that compute the following transformation:*

**Input:** *A number $Y$, between 0 and $2^m$ in Chinese Remainder Representation (CRR) using all primes $p < m^2$.*
**Output:** *The $m$ bits in the binary representation of $Y$.*

We will apply the Theorem for the case $m = 2^n + 1$. Thus $m^2 = (2^n + 1)^2$.

First, we observe any bit of the CRR of $val(X_w)$ can be computed in space polynomial in $n$. To obtain the bit $(p, j)$ we maintain for each variable in $X_w$ its residue modulo $p$ and update it according to the statements of $X_w$. The residue is a number less than $2^{2n+2}$ and hence requires $O(n)$ space, and the calculation for each statement can be done in $O(n)$ time. We also know that the statements of $f(w)$ can be generated in polynomial space. So altogether any bit in the CRR of $val(X_w)$ can be computed in PSPACE.

The idea now is to use the circuit $D_{2^{n+1}}$ to translate the CRR of $val(X_w)$ to its binary representation and show that any bit in the binary representation of $val(X_w)$ can be calculated in polynomial space. We show this by showing that the output of any gate of $D_{2^n+1}$ (when the input is CRR of $val(X_w)$) can be computed in polynomial space. This is done by inducting on the height of the gates present in the circuit. For the base case, we have seen that the input gates can be computed in PSPACE. For the inductive step, consider any gate $G$ at height $h + 1$. We can iterate over all gates $G'$ of $D_{2^{n+1}}$ and identify if $G'$ is a child of $G$ in polynomial space because the circuit $D_{2^n+1}$ is from a Dlogtime-uniform family. The children (of which there can be exponentially many) are all at height $h$ or less and hence output of each one can be computed in PSPACE. If $G$ is a NOT gate, output of $G$ can be easily computed from its child. If $G$ is a majority gate, we need only compare the number of children which evaluate to 1 against the number of those that evaluate to 0 which can be done in polynomial space by maintaining these numbers in binary counters.

In summary, to determine if $val(f(w)) > 0 \in L(f)$ we need to identify the value of two bits in the binary representation of $val(X_w)$ (one most significant bit and some other guessed bit). Even though $f(w)$ (and hence $X_w$) can be exponentially long, the good $\mathsf{SLP}$ conditions ensure that we can calculate the residues of the result in polynomial space. Using this along with Theorem 3 as above gives us the complete proof of our lemma.