On the Compositionality of Dynamic Leakage and Its Application to the Quantification Problem

Bao Trung Chu, Kenji Hashimoto, Hiroyuki Seki
Nagoya University, Japan
Email: trungchubao@sqlab.jp, {k-hasimt, seki}@i.nagoya-u.ac.jp

***Abstract*—Quantitative Information Flow (QIF), as summed up by Smith (2019), is traditionally defined as the expected value of information leakage over all feasible program runs. The traditional QIF fails to identify vulnerable programs where only a limited number of runs leak large amount of information. As discussed in Bielova (2016), a good notion for dynamic leakage and an efficient way of computing the leakage are needed. To address this problem, the authors have already proposed two notions for dynamic leakage and a method of quantifying dynamic leakage based on model counting. Inspired by the work of Kawamoto et al. (2017), this paper proposes two efficient methods for computing dynamic leakage, a compositional method along with the sequential structure of a program and a parallel computation based on the disjoint value domain decomposition. For the former, we investigate both exact and approximated calculations. For implementation, we utilize Binary Decision Diagrams (BDDs) and deterministic Decomposable Negation Normal Forms (d-DNNFs) to represent Boolean formulas in model counting. Finally, we show experimental results on several examples.**

Keywords–Dynamic leakage; Composition; Quantitative Information Flow; BDD; d-DNNF.

I. INTRODUCTION

Since first coined by [15] in 1982, the noninterference property has become one of the main criteria for software security [1][5]. A program is said to satisfy noninterference if any change in confidential information does not affect a publicly observable output of that program. However, noninterference is so strict that it blocks many useful, yet practically safe systems and protocols, such as password checkers, anonymous voting protocols, recommendation systems and so forth. Quantitative Information Flow (QIF) was introduced to loosen the security criterion in the sense that, instead of seeking *if there is* a case that a confidential input affects a public output, computing *how large* that effect is. That is, if the QIF of a program is insignificant, the program is still judged as secure. Because of its flexibility, QIF has gained much attention in recent years. However, it has an inherent shortcoming, as shown in the example below.

Example 1.1: Consider the following program taken from [10].

if $source < 16$ then $output \leftarrow 8 + source$
else $output \leftarrow 8$

Assume $source$ to be a non-negative 32-bits integer which is uniformly distributed on that domain. Then, there are 16 possible values of $output$, ranging from 8 to 23. Observing any number between 9 and 23 as an output reveals everything about the confidential $source$, whilst observing 8 leaks small information; there are many possible values of $source$ $(0, 16, 17, 18, \ldots, 2^{32}-1)$, which produce 8 as the output. QIF is defined as the average of the leakage over all possible cases. So, it fails to capture the above situation because we cannot distinguish *vulnerable* and *secure* cases if we take the average. Hence, as argued in [4], a notion for dynamic leakage should reflect individual leakage caused by observing an output.

As illustrated in Figure 1, there are two different scenarios of quantifying dynamic leakage. We call the first scenario, which corresponds to diagram (A), ***Compute-on-Demand*** (*CoD*), and the second, which corresponds to diagram (B), ***Construct-in-Advance*** (*CiA*). A box surrounded by bold lines represents a heavy-weighted process, which requires extensive computing resources. The main difference between (A) and (B) is the relative position of the heavy-weighted process, i.e., in (A), the process is put after augmenting an observed output and then the process is run each time we need (on demand) to compute dynamic leakage, or, in (B) the process is put before augmenting an observed output (in advance) so that we run the process only once for one program. In *CoD*, the heavy-weighted process is a projected model counting, for which off-the-shelf tools, such as SharpCDCL [32], DSharp-p [27] and GPMC [28] can be used. In *CiA*, the heavy-weighted process is the one that generates BDD [22] or d-DNNF [13], which are data structures to represent Boolean formulas given in Conjunctive Normal Form (CNF). Generally, it takes time to generate BDD or d-DNNF but counting all solutions (models) by using them is easy. *CiA* takes full advantage of this characteristic. Consider again Example 1.1 above. The set of all feasible pairs of ($source$, $output$) is 2^{32}. Even for such a simple program, using BDD or d-DNNF to store all those pairs is quite daunting in terms of both memory space and speed. Therefore, for programs with simple structure but a large number of input and output pairs, *CoD* works better than *CiA*. On the other hand, *CiA* is preferable to *CoD* when quantifying dynamic leakage is required many times on the same program. However, *CoD* or *CiA* alone is not a solution to the problem of scalability.

In this paper, we introduce two compositional methods for computing dynamic leakage inspired by the work of Kawamoto et al. [16] on the compositionality of static leakage. One method is to utilize the sequential structure of a given program $P = P_1; P_2$. We first analyze P_2 and then compute the leakage of P by analyzing P_1 based on the result on P_2. For the sequential composition, besides the benign yet time-consuming exact counting based on Breadth-First-Search (BFS), we also investigate an approximated approach. For an upper bound of the count, we leverage the results on each sub-program by Max#SAT in [14]. For a lower bound of the count, we simply use Depth-First-Search (DFS) with timeout, i.e., DFS will stop when the execution time exceeds the predetermined timeout.

The other method we propose is based on the decomposi-

 ISBN: 978-1-61208-746-7

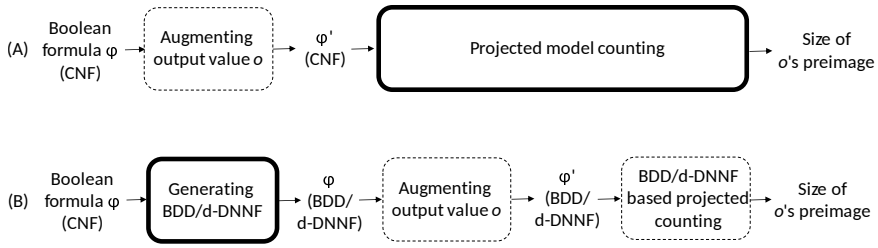

Figure 1. (A): Compute-on-Demand, (B): Construct-in-Advance.

tion of the value domain of a program. For example, we divide the input domain as $I = I_1 \cup I_2$ and the output domain as $O = O_1 \cup O_2$ of a program $P(I, O)$, compute the leakages of $P(I_i, O_j)$ for $i = 1, 2$ and $j = 1, 2$, then use them to compute the leakage of the whole program $P(I, O)$. This value domain based decomposition has two merits. First, it is flexible yet simple to adjust the components. Secondly, the exact dynamic leakage of the composed program can be simply derived by taking the sum of those of its components. Despite the fact that the number of components can be large, this approach is promising with parallel computing.

In summary, the contributions of this research are four-fold:

- We propose a compositional method for dynamic leakage computation based on the sequential structure inside a given program and the composability of the leakage of the whole program from those of subprograms.
- We propose another compositional method based on value domains, which is suitable for parallel computing.
- We propose an approximated approach where we upper bound the count using Max#SAT problem and lower bound the count by DFS with predetermined timeout.
- We prototype a tool that can do parallel computation based on value domain decomposition and both exact counting and approximated counting for the sequential composition. By using the tool, we investigate feasibility and advantages of the proposed compositional methods for computing dynamic leakage of several examples. The tool can be accessed freely via [26]. We also consider to develop a more capable open source analyzer based on this prototype in the future.

Related work *Definitions of QIF:* Smith [21] gives a comprehensive summary on entropy-based QIF, such as Shannon entropy, guessing entropy and min entropy and compares them in various scenarios. Clarkson et al. [11], on the other hand, include the attacker's belief into their model. Alvim et al. [3] introduce a gain function to generalize information leakage by separating the probability distribution and the impact of individual information. *Computational Complexity:* Yasuoka and Terauchi [24] prove complexity on computing QIF, including PP-hardness of precisely quantifying QIF for loop-free Boolean programs. Chadha and Ummels [8] show that the QIF bounding problem of recursive Boolean programs is EXPTIME-complete. *Precise Calculation:* In [17], Klebanov et al. reduce the QIF calculation to #SAT problem projected on a specific set of variables. On the other hand, Phan et al. [20] reduce the QIF calculation to #SMT problem to leverage existing Satisfiability Modulo Theory (SMT) solvers. Recently, Val et al. [23] reported a SAT-based method that can scale to programs of 10,000 lines of code. *Approximated Calculation:* Approximation is a reasonable alternative for scalability. Köpf and Rybalchenko [18] propose approximated QIF computation by sandwiching the precise QIF with lower and upper bounds using randomization and abstraction, respectively, with a provable confidence. LeakWatch, by Chothia et al. [9], also gives an approximation with provable confidence by executing a program multiple times. Its descendant, called HyLeak [7], combines the randomization strategy of its ancestor with precise analysis. Biondi et al. [6] utilize ApproxMC2, which provides approximation on the number of models of a Boolean formula in CNF by Markov Chain Monte Carlo method. *Composition of QIF:* Another attempt to the scalability is to break the system down into smaller fragments. In [16], Kawamoto et al. introduce two parallel compositions: with distinct inputs and with shared inputs, and give theoretical bounds on the leakage of the main program using those of the constituted sub-programs. Though our research was motivated by [16], we focus on a sequential structure of a target program and a decomposition of the value domain of the program while [16] uses a parallel structure of the target. *Dynamic Leakage:* Bielova [4] discusses the importance of dynamic leakage and argues that any well-known QIF notion is not appropriate as a notion for dynamic leakage. Recently, we proposed two notions for dynamic leakage, QIF_1 and QIF_2 and gave some results on computational complexity, as well as a quantifying method based on model counting [10].

The rest of the paper is organized as follows. We review the definition of dynamic leakage and describe our program model in Section 2. Section 3 is dedicated to a method for computing dynamic leakage based on the sequential composition and also proposes approximation methods. Section 4 proposes a parallel computation method based on value domain decomposition. Section 5 evaluates the proposed compositional methods including the comparison of *CiA* vs. *CoD* and exact vs. approximated computation based on the experimental results. Then, the paper is concluded in Section 6.

II. Preliminaries

A. Dynamic leakage

The standard notion for static QIF is defined as the mutual information between random variables S for secret input and O for observable output:

$$\text{QIF} = H(S) - H(S|O) \tag{1}$$

where $H(S)$ is the entropy of S and $H(S|O)$ is the expected value of $H(S|o)$, i.e., $H(S|O) = \sum_{o \in \mathcal{O}} p(o) H(S|o)$, and

$H(S|o)$ is the conditional entropy of S when observing an output o. Shannon entropy and min-entropy are often used as the definition of entropy, and in either case, $H(S) - H(S|O) \geq 0$ always holds by definition.

In [4], the author discusses the appropriateness of the existing measures for dynamic QIF and points out their drawbacks, especially, each of these measures may become negative. For example, if we adopt $H(S) - H(S|o)$ as a measure of dynamic QIF, the measure may become negative depending on an observed output value o.

Let P be a program with secret input variable S and observable output variable O. For notational convenience, we identify the names of program variables with the corresponding random variables. Throughout the paper, we assume that a program always terminates. The syntax and semantics of programs assumed in this paper will be given in the next section. Hereafter, let $\mathcal{S}$ and $\mathcal{O}$ denote the finite sets of input values and output values, respectively. For $s \in \mathcal{S}$ and $o \in \mathcal{O}$, let $p_{SO}(s,o)$, $p_{O|S}(o|s)$, $p_{S|O}(s|o)$, $p_S(s)$, $p_O(o)$ denote the joint probability of $s \in \mathcal{S}$ and $o \in \mathcal{O}$, the conditional probability of $o \in \mathcal{O}$ given $s \in \mathcal{S}$ (the likelihood), the conditional probability of $s \in \mathcal{S}$ given $o \in \mathcal{O}$ (the posterior probability), the marginal probability of $s \in \mathcal{S}$ (the prior probability) and the marginal probability of $o \in \mathcal{O}$, respectively. We often omit the subscripts as $p(s,o)$ and $p(o|s)$ if they are clear from the context. By definition, $p(s,o) = p(s|o)p(o) = p(o|s)p(s)$, $p(o) = \sum_{s \in \mathcal{S}} p(s,o)$, $p(s) = \sum_{o \in \mathcal{O}} p(s,o)$.

We assume that (the source code of) P and the prior probability $p(s)$ ($s \in \mathcal{S}$) are known to an attacker. For $o \in \mathcal{O}$, let $\mathrm{pre}_P(o) = \{s \in \mathcal{S} \mid p(s|o) > 0\}$, which is called the preimage of o (by the program P).

Considering the discussions in the literature, we define new notions for dynamic QIF that satisfy the following requirements [10]:

- (R1) Dynamic QIF should always be non-negative because an attacker obtains some information (although sometimes very small or even zero) when he observes an output of the program.
- (R2) It is desirable that dynamic QIF is independent of a secret input $s \in \mathcal{S}$. Otherwise, the controller of the system may change the behavior for protection based on the estimated amount of the leakage that depends on s, which may be a side channel for an attacker.
- (R3) The new notion should be compatible with the existing notions when we restrict ourselves to special cases, such as deterministic programs, uniformly distributed inputs, and taking the expected value.

The first notion is the self-information of the secret inputs consistent with an observed output $o \in \mathcal{O}$. Equivalently, the attacker can narrow down the possible secret inputs after observing o to the preimage of o by the program. We consider the self-information of $s \in \mathcal{S}$ after the observation as the logarithm of the probability of s divided by the sum of the probabilities of the inputs in the preimage of o (see the upper part of Figure 2, where bold lines indicate the mapping between $\mathcal{S}$ and $\mathcal{O}$ that are taken into consideration when defining QIF_1/QIF_2).

$$\mathrm{QIF}_1^P(o) = -\log\Big(\sum_{s' \in \mathrm{pre}_P(o)} p(s')\Big). \tag{2}$$

The second notion is the self-information of the joint events $s' \in \mathcal{S}$ and an observed output $o \in \mathcal{O}$ (see the lower part of Figure 2). This is equal to the self-information of o.

$$\begin{aligned}\mathrm{QIF}_2^P(o) &= -\log\Big(\sum_{s' \in \mathcal{S}} p(s',o)\Big) = -\log p(o) \\ &= -\log p(s,o) + \log p(s|o).\end{aligned} \tag{3}$$

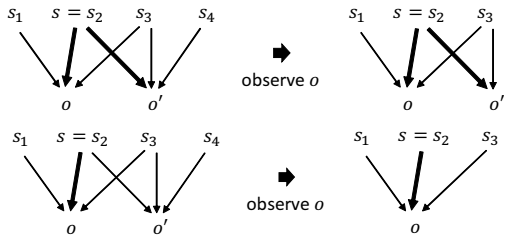

Figure 2. QIF_1 (the upper) and QIF_2 (the lower)

Both notions are defined by considering how much self-information values are reduced by observing an output. We propose these two notions because there is a trade-off between the easiness of calculation and the appropriateness [10].

Theorem 2.1 ([10]): If a program P is deterministic, for every $o \in \mathcal{O}$ and $s \in \mathcal{S}$,

$$\mathrm{QIF}_1^P(o) = \mathrm{QIF}_2^P(o) = -\log p(o).$$

If input values are uniformly distributed, $\mathrm{QIF}_1^P(o) = \log \frac{|\mathcal{S}|}{|\mathrm{pre}_P(o)|}$ for every $o \in \mathcal{O}$. □

B. Program model

We assume probabilistic programs where every variable stores a natural number and the syntactical constructs are assignment statement, conditional statement, probabilistic choice, while loop and concatenation:

$$\begin{aligned} b &::= \bot \mid \top \mid \neg b \mid b \vee b \mid e < e \\ e &::= X \mid n \mid e + e \\ c &::= \mathrm{skip} \mid X \leftarrow e \mid \mathrm{if}\ b\ \mathrm{then}\ c\ \mathrm{else}\ c\ \mathrm{end} \\ &\quad \mid c\ {}_r[]_{1-r}\ c \mid \mathrm{while}\ b\ \mathrm{do}\ c\ \mathrm{end} \mid c; c \end{aligned}$$

where $<, X, n, +$ stand for a binary relation on natural numbers, a program variable, a constant natural number and a binary operation on natural numbers, respectively, and r is a constant rational number representing the branching probability for a choice command where $0 \leq r \leq 1$. In the above BNFs, objects derived from the syntactical categories b, e and c are called conditions, expressions and commands, respectively. A command $X \leftarrow e$ assigns the value of expression e to variable X. A command $c_1\ {}_r[]_{1-r}\ c_2$ means that the program chooses c_1 with probability r and c_2 with probability $1 - r$. Note that this is the only probabilistic command. The semantics of the other constructs are defined in the usual way.

A program P has the following syntax:

$$P ::= \mathrm{in}\ \vec{S};\ \mathrm{out}\ \vec{O};\ \mathrm{local}\ \vec{Z}; c \mid P; P$$

where $\vec{S}, \vec{O}, \vec{Z}$ are sequences of variables which are disjoint from one another. A program is required to satisfy the following constraints on variables. We first define $In(P), Out(P), Local(P)$ for a program P as follows.

- If $P = \text{in } \vec{S}\text{; out } \vec{O}\text{; local } \vec{Z}; c$, we define $In(P) = \{V \mid V \text{ appears in } \vec{S}\}$, $Out(P) = \{V \mid V \text{ appears in } \vec{O}\}$ and $Local(P) = \{V \mid V \text{ appears in } \vec{Z}\}$. In this case, we say P is a simple program. We require that no varible in $In(P)$ appears on the left-hand side of an assignment command in P, i.e., no input variable is updated.
- If $P = P_1; P_2$, we define $In(P) = In(P_1)$, $Out(P) = Out(P_2)$ where we require that $In(P_2) = Out(P_1)$ holds. We also define $Local(P) = Local(P_1) \cup Local(P_2) \cup Out(P_1)$.

A program P is also written as $P(S, O)$ where S and O are enumerations of $In(P)$ and $Out(P)$, respectively. A program $P_1; P_2$ represents the sequential composition of P_1 and P_2. Note that the semantics of $P_1; P_2$ is defined in the same way as that of the concatenation of commands $c_1; c_2$ except that the input and output variables are not always shared by P_1 and P_2 in the sequential composition. If a program does not have a probabilistic choice, it is *deterministic*.

III. Sequential composition

This section proposes a method of computing both exact and approximated dynamic leakage by using sequential composition. For making the idea behind the proposed method understandable, we first assume the programs under analysis are deterministic with uniformly distributed input, so that the problem of quantifying dynamic leakage is reduced to model counting. Then, in the later part of this section, we will discuss the extensibility of the proposed method to probabilistic programs with input of an arbitrary distribution.

A. Exact calculation

For a program $P(S, O)$, an input value $s \in \mathcal{S}$ and a subset $\mathcal{S}'$ of input values, let

$$\begin{aligned} \text{post}_P(s) &= \{o \mid p(o|s) > 0\}, \\ \text{post}_P(\mathcal{S}') &= \bigcup_{s \in \mathcal{S}'} \text{post}_P(s). \end{aligned}$$

If P is deterministic and $\text{post}_P(s) = \{o\}$, we write $\text{post}_P(s) = o$.

Let $P = P_1; P_2$ be a program. We assume that $In(P_1), Out(P_1), In(P_2), Out(P_2)$ are all singleton sets for simplicity. This assumption does not lose generality; for example, if $In(P_1)$ contains more than one variables, we instead introduce a new input variable that stores the tuple consisting of a value of each variable in $In(P_1)$. Let $In(P) = In(P_1) = \{S\}$, $Out(P_1) = In(P_2) = \{T\}$, $Out(P) = Out(P_2) = \{O\}$, and let $\mathcal{S}, \mathcal{T}, \mathcal{O}$ be the corresponding sets of values, respectively. For a given $o \in \mathcal{O}$, $\text{pre}_P(o)$ and $p(o)$, which are needed to compute $\text{QIF}_1^P(o)$ and $\text{QIF}_2^P(o)$ (see (2) and (3)), can be represented in terms of those of P_1 and P_2 as follows.

$$\text{pre}_P(o) = \bigcup_{t \in (\text{pre}_{P_2}(o) \cap \text{post}_{P_1}(\mathcal{S}))} \text{pre}_{P_1}(t), \quad (4)$$

$$p(o) = \sum_{s \in \mathcal{S}, t \in \mathcal{T}} p(s) p_1(t|s) p_2(o|t). \quad (5)$$

```
1: Pre[2..n] ← empty
2: Stack ← empty
3: level ← n
4: acc_count ← 0
5: Push(Stack, o)
6: Pre[n] ← EnumeratePre(P_n, o)
7: while not Stack.empty and execution_time < timeout
   do
8:     if level = 1 then
9:         acc_count ← acc_count+ CntPre(P_1, Stack.top)
10:        level ← level + 1
11:        Pop(Stack)
12:    else
13:        v ← PickNotSelected(Pre[level])
14:        if v = AllSelected then
15:            level ← level + 1
16:            Pop(Stack)
17:        else
18:            Push(Stack, v)
19:            level ← level − 1
20:            if level > 1 then
21:                Pre[level] ← EnumeratePre(P_level, v)
22: return acc_count
```

Figure 3. LowerBound($P_1, \cdots, P_n, o$, timeout)

If $p(s)$ is given, we can compute (4) by enumerating $\text{pre}_{P_1}(t)$ for $t \in (\text{pre}_{P_2}(o) \cap \text{post}_{P_1}(\mathcal{S}))$ and also for (5). In practice, $\text{pre}_P(o)$ is computed by augmenting the information about an observed output value to the CNF that represents P, as illustrated in the flow of *CiA* and *CoD* in Figure 1. Then, the preimage can be either enumerated or counted, up to what is needed for the calculation. This approach can easily be generalized to the sequential composition of more than two programs, in which the enumeration is proceeded in a Breadth-First-Search fashion. However, in this approach, search space will often explode rapidly and lose the advantage of composition. Therefore, we come up with an approximation, which is explained in the next subsection, as an alternative.

B. Approximation

Let us assume that $P(S, O)$ is deterministic and S is uniformly distributed. In this subsection, we will derive both upper-bound and lower-bound of $|\text{pre}_P(o)|$, which provide lower-bound and upper-bound of $\text{QIF}_1^P(o) = \text{QIF}_2^P(o)$ respectively. In general, our method can be applied to the sequential composition of more than two sub-programs.

1) Lower bound: To infer a lower bound of $|\text{pre}_P(o)|$, we leverage Depth-First-Search (DFS) with a predefined timeout such that the algorithm will stop when the execution time exceeds the timeout and output the current result as the lower bound. The method is illustrated in Figure 3. For a program $P = P_1; P_2; \cdots; P_n$, an observable output o of the last subprogram P_n and a predetermined $timeout$, the algorithm in Figure 3 derives a lower bound of $|\text{pre}_P(o)|$ by those n subprograms.

In Figure 3, *CntPre(Q, o)* counts $|\text{pre}_Q(o)|$, *PickNotSelected(Pre[i])* selects an element of $Pre[i]$ that has not been traversed yet or returns *AllSelected* if there is no such element, and *EnumeratePre(P_i, v)* lists all elements in $\text{pre}_{P_i}(v)$. $Pre[i]$ stores $\text{pre}_{P_i}(o_i)$ for some o_i. For P_1, it is not necessary to store

1: $Result \leftarrow CntPre(P_n, o)$
2: **for** $i \leftarrow 1$ to n **do**
3: $\quad Result \leftarrow Result * MaxCount(P_i)$
4: **return** $Result$

Figure 4. UpperBound($P_1, \cdots, P_n, o$)

its preimage because we need only the size of the preimage. Lines 1 to 5 are for initialization. Line 6 enumerates $\text{pre}_{P_n}(o)$. Lines 7 to 21 constitute the main loop of the algorithm, which is stopped either when the counting is done or when time is up. When $level = 1$, lines 8 to 11 are executed and $CntPre$ will return $\text{pre}_{P_1}(Stack.top)$ in which $Stack.top$ is the input of P_2 that leads to output o of P_n, then back-propagate; lines 13 to 16 check if all elements in the preimage set of the current level is already considered and if so, back-propagate, otherwise push the next element onto the top of $Stack$ and go to the next level.

Theorem 3.1: In Figure 3, if $P_1, \cdots, P_n$ are deterministic, acc_count, which is returned at line 22, is a lower bound of the preimage size of o by $P_1, \cdots, P_n$. □

2) Upper bound: For an upper bound of $|\text{pre}_P(o)|$ we use Max#SAT problem [14], which is defined as follows.

Definition 3.1: Given a propositional formula $\varphi(X, Y, Z)$ over sets of variables X, Y and Z, the Max#SAT problem is to determine $max_X \#Y.\exists Z.\varphi(X, Y, Z)$.

If we consider a program Q, $In(Q)$, $Out(Q)$ and $Local(Q)$ as φ, Y, X and Z respectively, then, the solution X to the Max#SAT problem can be interpreted as the output value, which has the biggest size of its preimage set. In other words, $max_X \#Y.\exists Z.\varphi(X, Y, Z)$ is an upper bound of the size of pre_Q over all feasible outputs. Therefore, the product of those upper bounds of $|\text{pre}_{P_i}|$ over all i ($1 \leq i \leq n$) is obviously an upper bound of $|\text{pre}_P|$. The algorithm in Figure 4 computes this upper bound where $CntPre(P_n, o)$ returns the size of the preimage of o by P_n. Notice that, to avoid enumerating the preimages, which costs much computation time, we count only $|\text{pre}_{P_n}(o)|$. For $i = 1, ...n-1$, we compute $MaxCount(P_i)$ as an upper bound for pre_{P_i}, regardless of the corresponding output value. For prototyping, we used the tool developed by the authors of [14], which produces estimated bounds of Max#SAT with tunable confidence and precision. As explained in [14], the tool samples output values of k-fold self-composition of the original program. The greater k is, the more precise the estimation is, but also the more complicated the calculation of each sampling is. Note that $MaxCount(P_i)$ can be computed in advance only once. Though the precision is not always good in general, this approach provides a rather simple computation method. Also, note that the leakage is the logarithm of the model count.

Theorem 3.2: In Figure 4, if $P_1, \cdots, P_n$ are deterministic, then $Result$, which is returned at line 4, is an upper bound of the preimage size of o by $P_1, \cdots, P_n$. □

C. Extensibility to Probablistic Programs

When a program is probabilistic, a single input value may produce more than one output values. Therefore, when we count the preimage set of a specific output value, an input value may be counted multiple times, which results in an upper approximation. Though this does not invalidate our proposed algorithm, which computes an upper bound using Max#SAT, it could degrade the precision.

For the algorithms of exact counting and lower bounding, the following modifications will retain both their validity and precision as presented above when analyzing probabilistic programs. Notice that the algorithm in Figure 3 gives exact count when $timeout$ is sufficiently long.

- Maintain a list of distinct input values that produce the observed output value o'. In line 9 of Figure 3, instead of only counting models, enumerate all feasible input values, then update that list by adding the new input values. As the result, the final set is exactly the preimage set of o'.
- Provided the input distribution, i.e., the prior probability $p(s)$ of each input value s, QIF_1 can be calculated by taking the logarithm of the sum of $p(s)$ for s belonging to the preimage set, which is already computed by the above modification.
- Provided the channel matrix representing the conditional probability $p(o|s)$ of each output value o given an input value s, QIF_2 can be calculated by computing the probability $p(o')$ that the observed output value o' is produced.

IV. Value domain decomposition

Another effective method for computing the dynamic leakage in a compositional way is to decompose the sets of input values and output values into several subsets, compute the leakage for the subprograms restricted to those subsets, and compose the results to obtain the leakage of the whole program. The difference between the parallel composition in [16] and the proposed method is that in the former case, a program under analysis itself is divided into two subprograms that run in parallel, and in the latter case, the computation of dynamic leakage is conducted in parallel by decomposing the sets of input and output values.

Let $P(S, O)$ be a program. Assume that the sets of input values and output values, $\mathcal{S}$ and $\mathcal{O}$, are decomposed into mutually disjoint subsets as

$$\begin{aligned} \mathcal{S} &= \mathcal{S}_1 \uplus \cdots \uplus \mathcal{S}_k, \\ \mathcal{O} &= \mathcal{O}_1 \uplus \cdots \uplus \mathcal{O}_l. \end{aligned}$$

For $1 \leq i \leq k$ and $1 \leq j \leq l$, let P_{ij} be the program obtained from P by restricting the set of input values to $\mathcal{S}_i$ and the set of output values to $\mathcal{O}_j$ where if the output value o of P for an input value $s \in \mathcal{S}_i$ does not belong to $\mathcal{O}_j$, the output value of P_{ij} for input s is undefined. In practice, this disjoint decomposition can be done simply by augmenting the program under analysis with appropriate constraints on input and output.

By definition, for a given $o \in \mathcal{O}_j$,

$$\text{pre}_P(o) = \bigcup_{1 \leq i \leq k} \text{pre}_{P_{i,j}}(o). \tag{6}$$

By (2) and (3), we can compute QIF_1 and QIF_2 in a compositional way.

By Theorem 2.1, if P is deterministic and the prior probability of S is uniformly distributed, what we have to

compute is $|\text{pre}_P(o)|$, which can be obtained by summing up each $|\text{pre}_{P_{i,j}}(o)|$ by (6):

$$|\text{pre}_P(o)| \quad = \quad \sum_{1 \le i \le k} |\text{pre}_{P_{i,j}}(o)|. \tag{6}$$

Otherwise, probabilistic programs with arbitrary input distribution can be handled in a manner similar to the one described in the last paragraph of the previous section.

V. EXPERIMENTS

This section will investigate answers for the following questions: (1) How well does parallel computing based on the value domain decomposition improve the performance? (2) How well does sequential composition help improve the performance? (3) How well does approximation in the sequential composition work in terms of precision and speed? and (4) Is *CiA* always better than *CoD* or vice versa? We will examine those questions through a few examples: *Grade protocol* is for question (1), *Bit shuffle* and *Population count* are for (2) and (3), while (4) is considered based on both the former and the latter. The benchmarks and prototype are public in [26].

A. Setting up

The experiments were conducted on Intel(R) Xeon(R) CPU ES-1620 v3 @ 3.5GHz x 8 (4 cores x 2 threads), 32GB RAM, CentOS Linux 7. For parallel computation, we use OpenMP [12] library. At the very first phase, to transform C programs into CNFs, we leveraged the well-known CBMC [25]. For the construction of a BDD from a CNF and the model counting and enumeration of the constructed BDD, we use an off-the-shell tool PC2BDD [30]. We use PC2DDNNF [31] for the d-DNNF counterpart. Both of the tools are developed by one of the authors in another project. Besides, as the ordering of Boolean variables of a CNF greatly affects the BDD generation performance, we utilize FORCE [2] to optimize the ordering before transforming a CNF into a BDD. We use MaxCount [29] for estimating the answer of Max#SAT problem. We implemented a tool for algorithms in Figure 3 and Figure 4, as well as the exact count in sequential compositions in Java.

B. Grade protocol

This benchmark is taken from [19]. By this experiment, we investigated how well parallel computation improves the performance of counting models, hence of quantifying dynamic leakage, in value domain decomposition. This benchmark sums up (then takes the average of) the grades of a group of students without revealing the grade of each student. We used the benchmark with *4 students* and *5 grades*, and all variables are of 16 bits. For model counting, we suppose the observed output (the sum of students' grades) to be **1**, and hence the number of models is 4. GPMC [28], one of the fastest tools for quantifying dynamic leakage as shown in [10], was chosen as the representative tool for *CoD* approach. We manually decompose the original program into 4, 8 and 32 sub-programs by adding constraints on input and output of the program based on the value domain decomposition (the set of output values is divided into 2 and the set of input values is divided into 2, 4 or 16 disjoint subsets). Table I is divided into sub-divisions corresponding to specific tasks: BDD construction, d-DNNF construction and model counting based on different approaches. In each sub-division, the **bold number** represents the shortest execution time in each column (i.e., the same number of decomposed sub-programs, but different numbers of threads). '$-$' represents cases when the number of threads is greater than the number of sub-programs, which are obviously meaningless to do experiments.

TABLE I. TIME FOR CONSTRUCTING DATA STRUCTURE AND COUNTING MODEL.

		$n = 32$	$n = 8$	$n = 4$	$n = 1$
BDD Construction	$t = 32$	**218.53s**	–	–	–
	$t = 16$	222.27s	–	–	–
	$t = 8$	237.54s	**137.74s**	–	–
	$t = 4$	254.88s	144.55s	**155.90s**	–
	$t = 2$	376.21s	233.34s	214.65s	–
	$t = 1$	736.74s	450.85s	391.99s	**243.85**s
d-DNNF Construction	$t = 32$	93.17s	–	–	–
	$t = 16$	**91.49s**	–	–	–
	$t = 8$	107.31s	**123.48s**	–	–
	$t = 4$	141.27s	147.79s	**175.34s**	–
	$t = 2$	215.92s	226.93s	247.45s	–
	$t = 1$	398.99s	391.67s	457.38s	**304.88s**
Model Counting (*CiA* - BDD based)	$t = 32$	**0.21s**	–	–	–
	$t = 16$	0.22s	–	–	–
	$t = 8$	0.25s	**0.13s**	–	–
	$t = 4$	0.30s	0.16s	**0.16s**	–
	$t = 2$	0.65s	0.31s	0.24s	–
	$t = 1$	0.86s	0.36s	0.31s	**0.30s**
Model Counting (*CiA* - d-DNNF based)	$t = 32$	**0.05s**	–	–	–
	$t = 16$	**0.05s**	–	–	–
	$t = 8$	**0.05s**	**0.01s**	–	–
	$t = 4$	0.07s	**0.01s**	**0.01s**	–
	$t = 2$	0.12s	0.02s	0.02s	–
	$t = 1$	0.18s	0.04s	0.03s	**0.25s**
Model Counting (*CoD* - using GPMC)	$t = 1$	–	–	–	**44.69s**

In Table I, n: number of sub-programs decomposed from the original program; t: number of threads specified by *num_thread* compiling directive of OpenMP. Note that $n = 1$ means non-decomposition, $t = 1$ means a sequential execution and the number of physical CPUs is 8. From Table I, we can make the following inferences:

- As for the answer to question (1), parallel computing speeds up the calculation several times
  BDD Construction: **137.74s** vs. **243.85s**;
  d-DNNF Construction: **91.49s** vs. **304.88s**;
  Model Counting (CiA - BDD based): **0.13s** vs. **0.30s**. to tens times
  Model Counting (CiA - d-DNNF): **0.01s** vs. **0.25s**.
- In general, increasing the number of threads (up to the number of sub-programs) does improve the execution time in both the construction of BDD, d-DNNF and the model counting.
- When the number of sub-programs is close to the number of physical CPUs, which is eight, the execution time is among the best if not the best.

The performance with d-DNNF is better than that with BDD in this example, but this seems due to the implementation of the tools.

C. Bit shuffle and Population count

population_count is the 16-bit version of the benchmark of the same name given in [19]. In this experiment, the original program is decomposed into three sub-programs in such a way that each sub-program performs one bit operation

of the original. Inspired by *population_count*, we created the benchmark *bit_shuffle*, which consists of two steps: firstly it counts the number of bit-ones in a given secret number (by *population_count*, actually we took the count modulo 6 to increase the preimage size by the first part), then it shuffles those bits to produce an output value. This original program is divided into two sub-programs corresponding to the above-mentioned two steps. Though *bit_shuffle* is probabilistic (i.e., the shuffling part), the algorithm in Figure 3 still works, because there is always only one possible input value (i.e., the number of bit-ones) for an output of the sub-program corresponding to the latter step.

Table II shows execution times of constructing BDD and d-DNNF for two sample programs. The last three columns: *non-decompose*, *decompose (serial)* and *decompose (parallel)* are execution time when computed for the original program, computed sequentially and parallelly for the decomposed sub-programs, respectively. **Bold numbers** are the best execution times in those three.

For model counting, we let an output value be 3 (the number of models is 13110) for *bit_shuffle* and 7 (the number of models is 11440) for *population_count*. Table III presents the execution times for model counting where the underlined numbers are the exact counts, the **bold execution times** are the best results among approaches for the exact count of each benchmark and the *italic data* are of approximated calculations. The execution times for the lower bounds are predetermined timeouts, which were designed to be 1/2, 1/5 and 1/10 of the time needed by the exact count, followed by the time by *CoD*. In *bit_shuffle* benchmark, lower bounds based on d-DNNF were not improved (all are zero) even when the timeout was increased. This happened because an intermediate result of counting for one d-DNNF is unknown until the counting completes while this benchmark contains only two sub-programs and the size of the preimage by the second sub-program is always one (i.e., the number of times to count d-DNNFs is only two, one for the first sub-program and one for the second one).

TABLE II. BDD AND d-DNNF CONSTRUCTION TIME FOR DIFFERENT APPROACHES.

		non-decompose	decompose (serial)	decompose (parallel)
BDD Construction	bit_shuffle	>1 hour	33.90s	**33.46s**
	population_count	0.48s	0.66s	**0.40s**
d-DNNF Construction	bit_shuffle	424.64s	50.28s	**48.39s**
	population_count	1.19s	0.71s	**0.69s**

TABLE III. MODEL COUNTING: EXECUTION TIME AND THE CHANGING OF APPROXIMATION PRECISION.

			bit_shuffle		population_count	
CoD using GPMC (non-decompose)			0.49s	13110	**0.09s**	11440
CiA-BDD based (decompose)	Exact count		1.47s	13110	10.98s	11440
	Approximation	Lower bound	*0.75s*	*6243*	*5.5s*	*5776*
			0.30s	*1918*	*2.2s*	*888*
			0.15s	*574*	*1.1s*	*312*
			0.49s	*3713*	*0.09s*	*0*
		Upper bound	0.02s	14025	0.07s	5898240
CiA-d-DNNF based (decompose)	Exact count		**0.27s**	13110	3.50s	11440
	Approximation	Lower bound	*0.13s*	*0*	*1.75s*	*4712*
			0.05s	*0*	*0.70s*	*1314*
			0.03s	*0*	*0.35s*	*52*
			0.49s	*13110*	*0.09s*	*0*
		Upper bound	0.07s	14025	0.13s	5898240

From the experimental results, we obtain the following observations.

- As for the answer of question (2), in case of *bit_shuffle*, sequential composition helps speed up the construction of BDD more than 100 times (**33.46s** vs. $>$ 1 hour) and d-DNNF more than 8 times (**48.39s** vs. 424.64s). However, in case of *population_count*, the improvement is insignificant. For model counting, in both of the samples, sequential composition either helps just little or does not help.
- As for the answer of question (3), in case of upper bound for *bit_shuffle*, the precision is quite good. However, it was extremely low for *population_count*. For both of the samples, the calculation was very fast. For the lower bound, basically the precision gets better with longer timeout. In addition, because leakage is logarithm of model count, its upper bound and lower bound are much tighter than those of model count.

Note that, when we compare *CiA* with *CoD*, it is reasonable to ignore time of construction in *CiA*, because it happens only at the first time, then the result can be reused. For the question (4), in case of *grade protocol*, *CiA* shows a huge improvement over *CoD*, which is more than 4000 times (**0.01 s** vs. **44.69 s**). But in case of *population_count*, *CoD* is superior to *CiA* (**0.09 s** vs. 3.50 s). So, the answer is *NO*, i.e., sometimes *CiA* is better and the other time *CoD* is.

VI. CONCLUSION

In this paper, we focused on the efficient computation of dynamic leakage of a program and considered two approaches, *CoD* and *CiA*. Then, we proposed two compositional methods, namely, computation along with the sequential structure of the program and parallel computation based on value domain decomposition. In the first method, we also proposed approximations that give both lower bound and upper bound of model counting. Our experimental results showed that: (1) Parallel computation based on value domain decomposition works well generally; (2) Sequential composition sometimes helps significantly with construction of BDDs and d-DNNFs; (3) The precision of upper bounds depends on the way of decomposition while that of lower bound depends on the preset timeout; and (4) Both *CiA* and *CoD* are important because sometimes the former works better and the other times does the latter. All decomposition in the experiments were done manually. So, it is important to find a systematical way of deciding and guiding to a good decomposition. This is left as future work. One promising direction is to utilize static analysis, such as symbolic execution and program invariant. On the other hand, both BDD and d-DNNF have many applications other than computing dynamic leakage, but there is still a bottle neck at constructing them. The approach in this paper, *composition based on value domains*, can be a hint to speed up that process.

REFERENCES

[1] P. S. Aldous, *"Noninterference in expressive low-level languages"*, http://www.cs.utah.edu/~peteya/papers/diss.pdf [retrieved: October, 2019], PhD thesis, The University of Utah, 2017.

[2] F. Aloul, I. Markov, and K. Sakallah, *"FORCE: a fast and easy-to-implement variable-ordering heuristic"*, Great Lakes Symposium on VLSI (GLSVLSI), 2003, pp. 116–119.

[3] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith, *"Measuring information leakage using generalized gain functions"*, 21st Computer Security Foundations Symposium (CSF), 2012, pp. 265–279.

[4] N. Bielova, *"Dynamic leakage - a need for a new quantitative information flow measure"*, ACM Workshop on Programming Languages and Analysis for Security (PLAS), 2016, pp. 83–88.

[5] N. Bielova and R. Rezk, *"A taxonomy of information flow monitors"*, 5th International Conference on Principles of Security and Trust (POST), 2016, pp. 46–67.

[6] F. Biondi et al., *"Scalable approximation of quantitative information flow in programs"*, Verification, Model Checking, and Abstract Interpretation (VMCAI), 2018, pp. 71–93.

[7] F. Biondi, Y. Kawamoto, A. Legay, and L. M. Traonouez, *"HyLeak: hybrid analysis tool for information leakage"*, Automated Technology for Verification and Analysis (ATVA), 2017, pp. 156–163.

[8] R. Chadha and M. Ummels, *"The complexity of quantitative information flow in recursive programs"*, Research Report LSV-2012-15, Laboratoire Spécification & Vérification, École Normale Supérieure de Cachan, 2012.

[9] T. Chothia, Y. Kawamoto, and C. Novakovic, *"LeakWatch: estimating information leakage from Java programs"*, 19th European Symposium on Research in Computer Security (ESORICS), 2014, pp. 219–236.

[10] B. T. Chu, K. Hashimoto, and H. Seki, *"Quantifying dynamic leakage: complexity analysis and model counting-based calculation"*, IEICE Transactions on Information and Systems, Vol. E102-D, No. 10, Oct. 2019 (to appear), pre-print version: https://arxiv.org/abs/1903.03802.

[11] M. R. Clarkson, A. C. Myers, and F. B. Schneider, *"Quantifying information flow with beliefs"*, 18th Computer Security Foundations Symposium (CSF), 2009, pp. 655–701.

[12] L. Dagum and R. Menon, *"OpenMP: an industry-standard API for shared-memory programming"*, IEEE Computational Science & Engineering, Volume 5 Issue 1, Jan 1998, pp. 46–55.

[13] A. Darwiche, *"On the tractability of counting theory models and its application to belief revision and truth maintenance"*, Jounal of Applied Non-Classical Logics 11(1-2), 2001, pp. 11–34.

[14] D. J. Fremont, M. N. Rabe, and S. A. Seshia, *"Maximum Model Counting"*, AAAI Conference on Artificial Intelligence, 2017, pp. 3885–3892.

[15] J. A. Goguen and J. Meseguer, *"Security policies and security models"*, IEEE Symposium on Security and Privacy (S&P), 1982, pp. 11–20.

[16] Y. Kawamoto, K. Chatzikokolakis, and C. Palamidessi, *"On the compositionality of quantitative information flow"*, Logical Methods in Computer Science, Vol. 13(3:11) 2017, pp. 1–31.

[17] V. Klebanov, N. Manthey, and C. Muise, *"SAT-based analysis and quantification of information flow in programs"*, Quantitative Evaluation of Systems (QEST), 2013, pp. 177-192.

[18] B. Köpf and A. Rybalchenko, *"Approximation and randomization for quantitative information flow analysis"*, 23rd Computer Security Foundations Symposium (CSF), 2010, pp. 3–14.

[19] Q. S. Phan, *"Model counting modulo theories"*, PhD thesis, Queen Mary University of London, 2015.

[20] Q. S. Phan and P. Malacaria, *"All-solution satisfiability modulo theories: applications, algorithms and benchmarks"*, 10th International Conference on Availability, Reliability and Security (ARES), 2015, pp. 100–109.

[21] G. Smith, *"On the foundations of quantitative information flow"*, 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS), 2009, pp. 288–302.

[22] F. Somenzi, *"Binary decision diagrams"*, http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/reading/somenzi99bdd.pdf, 1999.

[23] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu, *"Precisely measuring quantitative information flow: 10k lines of code and beyond"*, IEEE European Symposium on Security and Privacy (EuroS&P), 2016, pp. 31–46.

[24] H. Yasuoka and T. Terauchi, *"On bounding problems of quantitative information flow"*, Journal of Computer Security (JCS), Vol. 19, 2011 November, pp. 1029–1082.

[25] C Bounded Model Checker, https://www.cprover.org/cbmc [retrieved: September, 2019].

[26] Dynamic Leakage Analyzer, https://bitbucket.org/trungchubao-nu/dla-composition/src/master/dla-composition/ [retrieved: September, 2019].

[27] DSharp-p, https://formal.iti.kit.edu/~klebanov/software/ [retrieved: September, 2019].

[28] GPMC, https://www.trs.css.i.nagoya-u.ac.jp/~k-hasimt/tools/gpmc.html [retrieved: September, 2019].

[29] MaxCount, https://github.com/dfremont/maxcount [retrieved: September, 2019].

[30] PC2BDD, https://git.trs.css.i.nagoya-u.ac.jp/t_isogai/cnf2bdd [retrieved: September, 2019].

[31] PC2DDNNF, https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/gpmc-dnnf [retrieved: September, 2019].

[32] SharpCDCL, http://tools.computational-logic.org/content/sharpCDCL.php [retrieved: September, 2019].

 ISBN: 978-1-61208-746-7