On the decidability of the information flow in Datalog using Magic Sets

[1]Antoun Yaacoub, [2]Hassan Safwan
[1]Professor, [2]PhD Student
[1]Applied Mathematics Department
[1]Faculty of Science, Lebanese University, Lebanon

***Abstract*—Information flow in logic programming was recently defined using the bottom-up evaluation approach. In this paper, we tackle the same question using the Magic Sets bottom-up evaluation approach. We will show, using this technique that the existence of the information flow is decidable. Finally, we will study the computational complexity of the decidability of the existence of the flow and prove that it is EXPTIME-complete.**

***Index Terms*—Logic programming, Information flow, Datalog, Decision problem, Computational complexity, Deductive database.**

I. INTRODUCTION

Computer technology has transformed the way people learn, work, and play. Computers have become an integral part of our everyday existence. They are used to store and to send personal letters, bank transactions, and highly sensitive military documents. For this, particular attention should be devoted to data security issues. In fact, data security refers to protective measures that are applied to prevent unauthorized access, disclosure and modification to data stored on computers, databases and websites. Controlling the flow of information is one of the main aspects of data security. Information flow describes controls that regulate the disclosure of information; controls to prevent leaking confidential data.

Information flow was introduced and defined for imperative programming by [1, 2, 3, 4, 5, 6, 7]. In fact, information flows from an item x to an item y, if after applying a sequence of command, the information initially in x affects "more or less" the information in y.

For example, the sequence $tmp \leftarrow x; y \leftarrow tmp;$ has information flowing from x to y because the value of x at the beginning of the sequence is revealed when the value of y is determined at the end of the sequence.

Logic programming which was developed in the early 1970s is considered as a declarative method for knowledge representation and programming. It is thus well-suited for both representing data and describing desired outputs.

Information flow in logic programming using top-down evaluation approaches was introduced in [8, 9, 10, 11, 12]. Yaacoub *et al.* defined the information flow in logic programming and developed mechanisms to detect such flows and proposed the notion of indistinguishability of flow and elaborated definitions of protection mechanisms, secure mechanisms, precise mechanisms and confidentiality policies based on this notion. They gave a secure and precise protection mechanism that prohibits any undesirable inferences and minimizes the number of denials of legitimate actions. The main results of their researches have shown that, it is undecidable to check whether there is a flow of information or not in the general setting of logic programs. But, if they restricted the language to Datalog programs using top-down evaluation, they achieved the decidability of the flow. They also worked on hierarchical, single variable occurrence (svo), non variable introducing (nvi) Datalog programs.

Yaacoub *et al.* [13] introduced the concept of bisimulation between Datalog goals with negation: two Datalog goals are bisimilar with respect to a given program when their SLDNFtrees are bisimilar. As proved, when the given logic program is stratified or restricted with negation, the problem of deciding whether two given goals are bisimilar is decidable. The proof of decidability of bisimulation problem for restricted logic program with negation is based on techniques that were developed for detecting loops in logic programming.

In [14], a newest approach of information flow detection in Datalog using bottom-up evaluation techniques was presented. It was shown that information flow detection in logic programming using the Very Naive and Semi Naive evaluation techniques is decidable and it is EXPTIME-Complete.

In this paper, we extend the work done in [14] by detecting information flow using the bottom-up Magic Sets evaluation technique in Datalog logic programming. In fact, Magic Sets is an efficient algorithm for bottom-up evaluation for Dalaog goals, since it is a query based algorithm.

In other words, it returns the same answer when the same query is run using a top-down evaluation technique.

In section 2, we briefly discuss the syntax and semantics and highlight information flow detection mechanisms and complexity results based on bottom-up evaluation techniques in Datalog logic programming. At the end of this section, Magic Sets bottom-up evaluation technique in Datalog is presented.

In section 3, we study the decidability and computational complexity of flow based on bottom up Magic Sets evaluation technique.

II. FRAMEWORK

Datalog is a declarative logic [15] language in which each formula is a function-free Horn clause, and every variable in the head of a clause must appear in the body of the clause. Predicates that appear in the head of the clause are called intensional predicates(IDB), while those that appear only in the body of the clause are called extensional predicates(EDB) [16].

Example 2.1. Let P be a Datalog program with the following input:

$p(x;y) \leftarrow r(x;y);$
$r(a;b) \leftarrow;$
Let the goal be $\leftarrow p(x;y)$.
IDB predicate: {p} (it appears in the head of the clause).
EDB predicate: {r} (it appears in the body of the clause only).

II.1 Bottom-Up Evaluation Techniques in Datalog

Datalog has two evaluation approaches, top-down and bottom-up. In what follows, we will focus on bottom-up evaluation techniques. These techniques are not goal directed; it starts from known facts and tries to infer new facts and stops when no more new facts can be generated, and it always terminates. In the literature, three main algorithms [17] were extensively used and applied to the bottom up approach: Very Naive, Semi Naive and Magic Sets algorithms.

1. Very Naive algorithm starts from scratch i.e it sets the values of all IDB predicates to ∅. Then, it computes all the possible facts. However, it may infer irrelevant facts according to a certain query. Note that the query is considered only at the end of the algorithm. For example, if we have a goal: $\leftarrow p(a;y)$, the answer of this goal is returned as soon as the algorithm terminates (i.e. after computing all the possible facts).
2. Semi Naive Algorithms do less work than the Very Naive algorithm by not recomputing already generated facts. In fact, the Semi Naive Algorithm works in a different manner. Since the values of EDB predicates never change during the whole execution except for the IDB values, we can deduce that if at round *i*, we can get a new fact (let's say *(a;b)*), this means that we used a newly generated fact at round **i**-1 to infer *(a;b)* [17]. To do so, this algorithm introduces a new type of predicates called incremental predicates. The incremental predicates are those for handling the newly generated facts [17].
3. As Ullman *et al.* mentioned [17, 18, 19, 20, 21], if we have a certain query and applying the Magic Sets transformation in addition to the Semi Naive Algorithm, it will take no more time than applying the same query with a top down technique. In other words, we can combine the advantage of the top down and that of the bottom up (goal directed and avoiding infinite loops). Magic Sets is about transforming given rules with a given goal to another rules satisfying the needs of the desired goal, so that we restrict the evaluation process of the bottom up technique. This transformation is based on something called adornment of the goal.
   For instance, if we have *p(a;y)* be some goal; the adornment of this goal is *bf* (*b* means that the first argument is bound to a constant, while f means that the second argument is free). Free argument means that a variable exists not a constant. Consequently, the transformation takes into consideration the bound arguments of the goal (i.e. the arguments that have constants).
   Now let's show how the transformation of the rules is done.
   Let *p(a;y)* be some goal, this means that the first variable is bound and the second is free (i.e *bf*).
   a. For each IDB predicate *p*, add *m_p* the magic predicate of *p* and the arguments of *m_p* are the bound variables in *p*. In our case *m_p(x)* since *x* is bound to a constant.
   b. For each IDB predicate *p* in rule *r* with *k* sub goals, suppose that *p* is a sub goal of *r* at position *i* and *q* is a sub goal of *r* at the position *i*-1 (with $i \leq k$). Create *m_p* with arguments of the bound arguments of *q* and sub goals from 1 to *k*-1. For example: let rule *r* be $p(x;y) \leftarrow q(x; z), p(z;y)$, the new magic rule will be $m\_p(z) \leftarrow q(x; z)$. Since *z* is bound in *p*, so we take the *z* in *q*.
   c. Add a new magic fact corresponding to the original goal with the bound arguments: in our case we add *m_p(a)* since the original goal is *p(a;y)*.

Very Naive and Semi Naive algorithms can be found in [13].

Example 2.2. Let P be a Datalog program with the following input:

$q(a;b) \leftarrow;$
$q(b;c) \leftarrow;$
$q(c;d) \leftarrow;$
$p(x;y) \leftarrow q(x;y);$
$p(x;y) \leftarrow q(x; z), p(z;y);$

Let the goal be $\leftarrow p(a;y)$.
In this example, we have 3 facts **q(a;b), q(b;c)** and $q(c;d)$. Also, q is the only EDB predicate since it appears only in the body of the rules, whereas p is the only IDB predicate. Moreover, we have two rules as shown earlier.

Now let us execute this program using the Very-Naïve Bottom-up evaluation technique.
Initially, set all the IDB's to ∅, here we only have *p*.
So $P = \emptyset$, and $Q = \{(a;b); (b;c); (c;d)\}$
In round 1: $P = \{(a;b); (b;c); (c;d)\}$. Clearly, only the first rule works in the first round because *P* is ∅, so it takes all the tuples from *Q*.
In round 2: $P = \{(a;b); (b;c); (c;d); (a;c); (b;d)\}$. Here, the second rule also works. Applying the join between *p* and *q*, we get the newly added tuples. The algorithm continues because the values of *P* changed.
In round 3: $P = \{(a;b); (b;c); (c;d); (a;c); (b;d); (a;d)\}$. New facts are generated, so the algorithm continues.
In round 4: $P = \{(a;b); (b;c); (c;d); (a;c); (b;d); (a;d)\}$. No new facts are generated, the algorithm stops. Now it considers the query, $\pi_y(\sigma_{x=a}(P(x;y))$. The answer is $y = \{b;c;d\}$.

As noticed, in each round, it generated new facts. But, it recomputed previously generated facts. For example, in round 3, the new generated fact is *(a;d)*, but, *(a;b), (b;c), (c;d), (a;c)* and *(b;d)* are recomputed again. This algorithm is thus inefficient.

Now let us show the difference between the Very-Naïve and Semi-Naive algorithms. The difference occurs in the rules:
1. $\Delta p(x;y) \leftarrow q(x;y)$
2. $\Delta p(x;y) \leftarrow q(x; z); Dp(z;y)$
where $\Delta p(x;y)$ is an incremental predicate associated to the predicate $p(x;y)$ that only holds the tuples added in the previous round.

Now let's run this program. In the first round, the incremental predicates take values from the rules that have only EDB's in the body (here rule 1).
Thus, $\Delta P = \{(a;b); (b;c); (c;d)\}$ and *P* has the same values of ΔP.
In round 1: $\Delta P = \{(a;b); (b;c); (c;d); (a;c); (b;d)\}$
$\Delta P = \Delta P - P = \{(a;c); (b;d)\}$
$P = P \cup \Delta P = \{(a;b); (b;c); (c;d); (a;c); (b;d)\}$
Here we removed the newly added tuples, then we will add those tuples to the original predicate which is *P*. $\Delta P \neq \emptyset$, the algorithm continues.
In round 2: $\Delta P = \{ (a;c); (b;d); (a;d)\}$
$\Delta P = \Delta P - P = \{(a;d)\}$
$P = P \cup \Delta P = \{(a;b); (b;c); (c;d); (a;c); (b;d); (a;d)\}$
Notice the difference here, ΔP contains only two tuples (not 5 as in the Very Naive Algorithm, here is the optimization).
In round 3: $\Delta P = \{(a;d)\}$
$\Delta P = \Delta P - P = \emptyset$
$P = P \cup \Delta P = \{(a;b); (b;c); (c;d); (a;c); (b;d); (a;d) \}$
Consequently, the algorithm stops since all the incremental predicates became ∅. The answer of the query is: $y = \{b;c;d\}$. Same answer as before but more efficient since it didn't recompute already generated facts in each round.

Now we have distinguished the difference between the two algorithms. Next, we will cover one more optimized algorithm applied by the bottom up evaluation which is the Magic Sets transformation. We will show step by step how we achieve the desired answer of the query based on Magic Sets.

First rule: $p(x;y) \leftarrow q(x;y)$ will be transformed into $p(x;y) \leftarrow m\_p(x); q(x;y)$ (rule 1).
Second rule: $p(x;y) \leftarrow q(x; z); p(z;y)$ will be transformed into $p(x;y) \leftarrow m\_p(x); q(x; z); p(z;y)$ (rule 1).
We add a new magic rule $m\_p(z) \leftarrow m\_p(x); q(x; z)$ (rule 2)
Finally, we add the new magic fact $m\_p(a)$. Now using the Semi Naive Algorithm, we must get the same answer when the same program is executed using top down technique.
The first loop sets $\Delta m\_p = \{a\}$ and $m\_p = \{a\}$

Round	***ΔP***	***P***	***Δm_P***	***m_P***
0			$\Delta m\_P = \{a\}$	$m\_p = \{a\}$
1	$\Delta P = \{(a;b)\}$	$P = \{(a;b)\}$	$\Delta m\_P = \{a;b\}$ $\Delta m\_P = \{b\}$	$m\_p = \{a;b\}$

2	$\Delta P = \{(a;b)\}$ $\Delta P = \varnothing$	$P = \{(a;b)\}$	$\Delta m\_P = \{b;c\}$ $\Delta m\_P = \{c\}$	$m\_p = \{a;b;c\}$
3	$\Delta P = \varnothing$	$P = \{(a;b)\}$	$\Delta m\_P = \{c;d\}$ $\Delta m\_P = \{d\}$	$m\_p = \{a;b;c;d\}$
4	$\Delta P = \varnothing$	$P = \{(a;b)\}$	$\Delta m\_P = \{c;d\}$ $\Delta m\_P = \varnothing$	$m\_p = \{a;b;c;d\}$

As we see, the answer is in *m_p*, the same as the top down will return. Note that a in the answer is added via the magic fact.

More details about Magic Sets transformation can be found in [17, 18].

II.2 Information Flow Definitions using Bottom Up Evaluation Techniques

We will restate information flow definitions in Datalog programs using a bottom up evaluation techniques. Most of the information is this part can be found in [13].

The first definition is based on Success/Failure of the goals. Let *P* be a Datalog program and *G(x;y)* be a two variables goal. There is a flow of information from *x* to *y* in $P\left(x \overset{SF^P}{\rightarrow_G} y\right)$ based on SF in goal *G* and Program *P* iff there exists *a, b* $\in U_{L(P)}$ such that $P \cup \{G(a;y)\}$ succeeds and $P \cup \{G(b;y)\}$fails. This means that when the user only sees the outputs of computations in terms of successes and failures, ∃ two different *a, b* $\in U_{L(P)}$ such that the user can distinguish between the outputs of the goals without seeing what concerns *a* and *b*.

The second definition is based on the substitution answers of the goals. Let *P* be a Datalog program and *G(x;y)* be a goal. We can say that there is an information flow from *x* to *y* in *G(x;y)* with respect to substitution answers in P iff *a, b* $\in U_{L(P)}$ such that $\Theta(P \cup \{G(a;y)\}) \neq \Theta(P \cup \{G(a;y)\})$. In this definition, the user only sees the outputs of computations in terms of substitution answers. Consequently, there is a flow of information from *x* to *y* if this user can distinguish the output of $P \cup \{G(a;y)\}$ and the output of $P \cup \{G(b;y)\}$.

Note that:

- The existence of a flow with respect to substitution answers does not entail the existence of a flow with respect to success and failures.
- If $x \overset{SF^P}{\rightarrow_G} y$ based on success/failure in goal *G(x;y)* and program *P*, then $x \overset{SA^P}{\rightarrow_G} y$ based on substitution answers in goal *G(x;y)* and program *P*, for *P* a logic program and *G(x;y)* a two variables goal.

III. Decidability / Complexity

We study the computational complexity of the following decision problems:

$$\pi_{SF}\begin{cases} Input: \text{A logic program P, a two variables goal G(x; y)} \\ Output: \text{Determine whether } x \overset{SF^P}{\rightarrow_G} y \end{cases}$$

$$\pi_{SA}\begin{cases} Input: \text{A logic program P, a two variables goal G(x; y)} \\ Output: \text{Determine whether } x \overset{SA^P}{\rightarrow_G} y \end{cases}$$

As we mentioned before, the rules of the program are transformed for every query in order to restrict the evaluation process. As we will see in the upcoming algorithm, we will apply the algorithm on each constant of the Herbrand Universe.

In the following, the EVAL function is a combination of select, project and join (database techniques).

We now study the decidability and computational complexity of (π_{SF}) based on Bottom Up Evaluation Magic Sets Algorithm. For this, consider the following algorithm:

Algorithm 1: Magic Sets - Flow Detection based on Success/Failure

Require: Program P (

- List of EDB predicates $\{e_1; \ldots; e_m\}$
- List of IDB predicates $\{r_1; \ldots; r_k\}$and new IDB predicates are included in this list representing the magic predicates, for example: for each IDB predicate r_i, there exists m_r_i represents the magic predicate of r_i
- List of relations $\{E_1; \ldots; E_m\}$to serve as values of EDB predicates.
- List of incremental predicates $\{R_1; \ldots; R_k\}$where, for each IDB predicate P_i, there is an incremental predicate ΔP_i that holds only the tuples added in the previous round

- Distinct Herbrand Universe $U_{L(P)}=\{a_1; \ldots ; a_n\}$
- Goal: $\leftarrow g(x;y)$
- New fact representing the bound variable of the goal, for example: if Goal: $\leftarrow g(a;y)$ then the new fact is $m\_g(a)$.

)

Ensure: $x \overset{SF^P}{\rightarrow_G} y$

```
begin
    succeed = false
    for i = 1 to n do
        for i = 1 to k do
            ΔRi=EVAL(ri; E1; … ; Em; ∅; … ; ∅)
            if predicate(ΔRi,'g') and arity(ΔRi,2) then
                succeed = true
                remove(constant(ΔRi),UL)
            Ri = ΔRi
        repeat
            for i = 1 to k do
                ΔQi = ΔRi
            for i = 1 to k do
                ΔRi=EVAL(ri; E1; … ; Em; R1; … ; Rk; ΔQ1 ;… ; ΔQk)
                ΔRi = ΔRi - Ri
                if predicate(ΔRi,'g') and arity(ΔRi,2) then
                    succeed = true
                    remove(constant(ΔRi),UL)
            for i = 1 to k do
                Ri = Ri ∪ΔRi
        until ΔRi =∅ ∀i, 1 ≤ i ≤ k
    if succeed == true and UL contains at least one element then
        return true
    return false
end
```

In order to demonstrate the decidability of (π_{SF}), we need to prove the following lemmas:

Lemma 3.1 (*Termination*). Algorithm 1 terminates.

Lemma 3.2 (*Completeness*). If $x \overset{SF^P}{\rightarrow_G} y$, then Algorithm 1 returns `true`.

Lemma 3.3 (*Soundness*). If Algorithm 1 returns true, then $x \overset{SF^P}{\rightarrow_G} y$.

Proof of Lemma 3.1. According to Jeffrey D.Ullman [17], suppose we have an upper limit of arity, let's say a, and number of symbols, let it be b. So, we have b^a different tuples. Also, let m be the number of IDB predicates, so, this algorithm needs at most mb^a rounds to reach a fixed point and terminates.

□

Proof of Lemma 3.2. If $x \overset{SF^P}{\rightarrow_G} y$, thus there ∃ two constants a, $b \in U_L$ with a $a \neq b$ such that $G(a;y)$ succeeds whereas $G(b;y)$ fails. According to this algorithm, constant a will be removed from U_L and succeed will be set to true, while b will remain in U_L. Consequently, the if condition will be true and the algorithm will return true.

On the other hand, if there is no flow $x \overset{SF^P}{\rightarrow_G} y$, we have two cases:

(1) ∀ constant $a_i \in U_L (1 \leq i \leq n)$, $G(a_i;y)$ fails, OR

(2) ∀ constant $a_i \in U_L (1 \leq i \leq n)$, $G(a_i;y)$ succeed and a_i will be removed from U_L.

In both cases, this will lead to the falseness of the `if` condition and the algorithm will return `false`.

□

Proof of Lemma 3.3. If the algorithm returns `true`, thus the `if` condition is true, consequently `succeed = true` (1) and U_L L contains at least one element (2).

From (1), there ∃ constant $a \in U_L$ such that $G(a;y)$ succeeded and a is removed from U_L.

From (2), there ∃ another constant $b \neq a \in U_L$ such that $G(b;y)$ failed. ($b \neq a$ because U_L is distinct).

Thus, $x \overset{SF^P}{\rightarrow_G} y$.

On the other hand, if the algorithm returns `false`, we have two cases:

(1) `succeed = false` which leads to the fact that ∀ constant $a_i \in U_L$ $(1 \leq i \leq n)$, $G(a_i;y)$ failed.

(2) U_L contains zero elements, which means that ∀ constant $a_i \in U_L$ $(1 \leq i \leq n)$, $G(a_i;y)$ succeeded and a_i removed from U_L.

In both cases ∄ two constants $a;b \in U_L$ with $a \neq b$ such that $G(a;y)$ succeeds and $G(b;y)$ fails. Thus, there is no flow.

□

As a consequence of lemmas 3.1 – 3.3, we have:

Theorem 3.4. Algorithm 1 is a sound and complete decision procedure for (π_{SF}).

It follows that (π_{SF}) is decidable. Moreover,

Theorem 3.5. (π_{SF}) is EXPTIME-Complete.

Proof. (*Membership*) Now, we will explain briefly about the complexity of this algorithm by estimating the time needed for the loops and certain instructions.

```
begin
    for i = 1 to k do
          Ri = ∅
end
```

This for loop needs about $\theta(k)$ as a time complexity.
EVAL function is a database technique ($\sigma;\pi;\bowtie$).
Suppose we have a table of length k.
$\sigma : \theta(k)$
$\pi : \theta(k)$
Suppose we have two tables of length k, l respectively.
$\bowtie: \theta(k * l)$
Also, we have "`repeat - until`" and an inner `for` loop, so, this algorithm can be executed in EXPTIME in the worst case.

(*Hardness*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard [11, 22]:

$$\pi \begin{cases} Input: \text{A logic program P, a ground atom A} \\ Output: \text{P} \cup \text{A (A is a logical consequence of P)} \end{cases}$$

Let $(P;A)$ be an instance of π and let $(P';g(x;y))$ be the instance of π_{SF} defined by $P' = P \cup \{g(a;y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P \cup A$ iff $x \overset{SF^{P'}}{\rightarrow}_G y$ based on success/failure in goal g and program P'.

($\rightarrow$) Suppose that A is a logical consequence of P, thus, $P' \cup \{\leftarrow g(a;y)\}$succeeds and $P' \cup \{\leftarrow g(b;y)\}$ fails. Consequently, $x \overset{SF^{P'}}{\rightarrow}_G y$ based on success/failure in goal g and program P'.

($\leftarrow$) Suppose that $x \overset{SF^{P'}}{\rightarrow}_G y$ based on success/failure in goal g and program P'. Then $\exists\, a', b' \in U_L(P)$ such that $P' \cup \{\leftarrow g(a';y)\}$ succeeds and $P' \cup \{\leftarrow g(b';y)\}$fails. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P \cup A$.

□

We now study the decidability and computational complexity of (π_{SA}) based on Bottom Up Evaluation Magic Sets Algorithm. For this, consider the following algorithm:

Algorithm 2: Magic Sets - Flow Detection based on Substitution Answers

Require: Program P (

- List of EDB predicates $\{e_1; \ldots ; e_m\}$
- List of IDB predicates $\{r_1; \ldots ; r_k\}$and new IDB predicates are included in this list representing the magic predicates, for example: for each IDB predicate r_i, there exists $m\_ r_i$ represents the magic predicate of r_i
- List of relations $\{E_1; \ldots ; E_m\}$to serve as values of EDB predicates.
- List of incremental predicates $\{R_1; \ldots ; R_k\}$where, for each IDB predicate P_i, there is an incremental predicate ΔP_i that holds only the tuples added in the previous round
- Distinct Herbrand Universe $U_{L(P)}=\{a_1; \ldots ; a_n\}$
- Goal: $\leftarrow g(x;y)$
- New fact representing the bound variable of the goal, for example: if Goal: $\leftarrow g(a;y)$ then the new fact is $m\_g(a)$.

)

Ensure: $x \overset{SA^{P}}{\rightarrow}_G y$
begin
    for i = 1 to n **do**

```
            for i = 1 to k do
                    ΔRi=EVAL(ri; E1; … ; Em; ∅; … ; ∅)
                    if predicate(ΔRi,'g') and arity(ΔRi,2) then
                            A[constant(ΔRi1)] = constant(ΔRi2)
                            remove(constant(ΔRi),UL)
                    Ri = ΔRi
            repeat
                    for i = 1 to k do
                            ΔQi = ΔRi
                    for i = 1 to k do
                            ΔRi=EVAL(ri; E1; … ; Em; R1; … ; Rk; ΔQ1 ;… ; ΔQk)
                            ΔRi = ΔRi - Ri
                            if predicate(ΔRi,'g') and arity(ΔRi,2) then
                                    A[constant(ΔRi1)] = constant(ΔRi2)
                                    remove(constant(ΔRi),UL)
                    for i = 1 to k do
                            Ri = Ri ∪ΔRi
            until ΔRi =∅ ∀i, 1 ≤ i ≤  k
    if A is empty or (UL contains zero elements not in keys of A and
                    A doesn't contain two different substitution answers)then
            return true
    return false
end
```

Where, A is a set of (key/value) pairs representing the substitution answers of each of the constants in U_L. For example, if we have $A[a] = \{a;b;c\}$ this means that a is the key and $\{a;b;c\}$ is the value representing the substitution answer of a.

In order to demonstrate the decidability of (π_{SA}), we need to prove the following lemmas:

Lemma 3.6 (*Termination*). Algorithm 2 terminates.

Lemma 3.7 (*Completeness*). If $x \overset{SA^P}{\rightarrow}_G y$, then Algorithm 2 returns `true`.

Lemma 3.8 (*Soundness*). If Algorithm 2 returns true, then $x \overset{SA^P}{\rightarrow}_G y$.

Proof of Lemma 3.6. Same proof as 3.1.

□

Proof of Lemma 3.7. If $x \overset{SA^P}{\rightarrow}_G y$, thus there ∃ constants $a, b \in U_L$ with a $a \neq b$ such that $\Theta(g(a;y)) \neq \Theta(g(b;y))$. So, according to this algorithm, we have two cases:

(1) one of the substitution answers is ∅, this means that this constant is not removed from U_L, while the other one has a non empty set and removed from U_L and added to A (A now is not empty). Thus, A is not empty and U_L contains at least one element not in keys of U_L. So, the `if` condition will be false, consequently this algorithm will return `true`.

(2) All the constants are removed from U_L. This means that U_L contains zero elements not in keys of A. Also, A is not empty and contains two different substitution answers. Consequently, the `if` condition is false, thus, the algorithm will return `true`.

On the other hand, if there is no flow $x \overset{SA^P}{\rightarrow}_G y$, we have two cases:

(1) ∀ constant $a_i \in U_L (1 \leq i \leq n)$, $\Theta(G(a_i;y)) = \emptyset$, OR

(2) ∀ constant $a_i \in U_L (1 \leq i \leq n)$, $\Theta(G(a_i;y)) = \Theta(G(a_j;y))$.

In both cases, this will lead to the truth of the `if` condition.

Consequently, the algorithm will return `false`.

□

Proof of Lemma 3.8. If the algorithm returns `true`, thus the `if` condition is false false. This means that A is not empty and (U_L contains at least one element not in keys of A or A contains two different substitution answers).

(1) (A is not empty and U_L contains at least one element not in keys of A), OR

(2) (A is not empty and A contains two different substitution answers).

In (1), A is not empty, then there ∃ a constant $a \in U_L$ such that $\Theta(G(a;y)) \neq \emptyset$, and U_L contains at least one element not in keys of A means, there ∃ another constant $b \neq a \in U_L$ such that $\Theta(G(b;y)) = \emptyset$. Consequently, $x \overset{SA^P}{\rightarrow}_G y$.

In (2) as in (1) for A is not empty. And A contains two different substitution answers means that there ∃ two constants $a;b \in U_L$ with $a \neq b$ such that $\Theta(G(a;y)) \neq \Theta(G(b;y))$. Thus, $x \overset{SA^P}{\rightarrow}_G y$.

On the other hand, if the algorithm returns `false`, we have two cases:

(1) A is empty. Therefore, $\forall$ constant $a_i, a_j \in U_L$ with $a_i \neq a_j$, we have $\Theta(G(a_i;y)) = \Theta(G(a_j;y)) = \varnothing$. Consequently, there is no flow.

(2) U_L contains zero elements not in keys of A and A doesn't contain two different substitution answers. This means that $\forall$ constant $a_i \in U_L$ $(1 \leq i \leq n)$; $\Theta(G(a_i;y)) \neq \varnothing$. But A doesn't contain two different substitution answers, so, $\forall$ constant $a_i, a_j \in U_L$ with $a_i \neq a_j$; $\Theta(G(a_i;y)) = \Theta(G(a_j;y))$. Thus, there is now flow.

□

As a consequence of lemmas 3.6 – 3.8, we have:

Theorem 3.9. Algorithm 2 is a sound and complete decision procedure for (π_{SA}).

It follows that (π_{SA}) is decidable. Moreover,

Theorem 3.10. (π_{SA}) is EXPTIME-Complete.

Proof. (*Membership*) Concerning the time complexity of this algorithm, our study will be based on the set of pairs A.
First, we have to sort the set of substitution answers which is called A using merge-sort algorithm:
worst case:
Suppose the length of keys of A is n. And each entry of A (i.e values of substitution answers) is n also.
To sort one row it is about $\theta(nlogn)$.
To sort all rows, time complexity will be about $\theta(n^2logn)$.
To search an element in a sorted set, complexity will be $\theta(log_2 n)$.
To compare all rows with each other $\theta(n^2logn)$.
Consequently, this algorithm is executed in EXPTIME.

(*Hardness*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard [11, 22]:

$$\pi \begin{cases} Input: \text{A logic program P, a ground atom A} \\ Output: \text{P} \cup \text{A (A is a logical consequence of P)} \end{cases}$$

Let $(P;A)$ be an instance of π and let $(P';g(x;y))$ be the instance of π_{SA} defined by $P' = P \cup \{g(a;y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P \cup A$ iff $x \overset{SA^{P'}}{\rightarrow}_G y$ based on substitution answers in goal g and program P'.

($\rightarrow$) Suppose that A is a logical consequence of P, thus, $\Theta(P' \cup \{\leftarrow g(a;y)\}) \neq \varnothing$ and , $\Theta(P' \cup \{\leftarrow g(b;y)\}) = \varnothing$. Consequently, $x \overset{SA^{P'}}{\rightarrow}_G y$ based on substitution answers in goal g and program P'.

($\leftarrow$) Suppose that $x \overset{SA^{P'}}{\rightarrow}_G y$ based on substitution answers in goal g and program P'. Then $\exists\, a', b' \in U_L(P)$ such that $\Theta(P' \cup \{\leftarrow g(a';y)\}) \neq \varnothing$ and $\Theta(P' \cup \{\leftarrow g(b';y)\}) = \varnothing$. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P \cup A$.

□

IV. Conclusion

In this paper, we continued the work presented in [14] and studied the decidability and computational complexity of the flow detection using the Magic Sets bottom-up evaluation technique.

We showed that information flow detection in logic programming using the aforementioned algorithm is decidable and it is EXPTIME-Complete.

Future work can be dedicated to the study of the third information flow definition in Datalog programs based on bottom up evaluation, which is the bisimilarity between goals.

Moreover, we can base our research on another type of Datalog programs such as stratified Datalog, which is a Datalog program that accepts negative literals in its rules, i.e. $p(x;y) \leftarrow \neg q(x;y)$.

References

[1] D. Bell, And L. Lapadula, "Secure Computer Systems: Mathematical Foundations And Model." The Mitre Corporation Bedford Ma Technical Report M74244 , Vol. 1, 1973.

[2] S. Foley, "A Model for Secure Information Flow." Security And Privacy, 1989. Proceedings., 1989 IEEE Symposium, P. 248-258, 1989.

[3] D. A Denning, "Lattice Model Of Secure Information Flow." Commun. Acm 19, 5, P. 236-243, 1976

[4] J. Goguen, And J. Meseguer, "Security Policies And Security Models." IEEE Symposium on Security and Privacy, P. 11-20, 1982.

[5] D. Denning, and P. Denning, "Certification of Programs for Secure Information Flow." Commun. ACM 20, 7, P. 504-513, 1977.

[6] J. Fenton, "Memoryless Subsystems ."The Computer Journal, 17, P. 143-147, 1974.

[7] D.Denning, "Cryptography and Data Security," Addison-Wesley, 1982.

[8] P. Balbiani, and A.Yaacoub, "Deciding the Bisimilarity Relation Between Datalog Goals (Regular Paper)." In European Conference on Logics in Artificial Intelligence (Jelia), Toulouse, L. Farinas Del Cerro, A. Herzig, And J. Mengin, Eds., Springer, Pp. 67–79, 2012.

[9] A. Yaacoub, "Towards an Information Flow in Logic Programming." International Journal of Computer Science Issues (Ijcsi) 9, 2, 2012.

[10] A. Yaacoub, and A. Awada, "Inference Control on Information Flow in Logic Programming." International Journal of Computer Science: Theory and Application (Ijcsta), Vol. 3, Issue.: 1, P. 13-22, 2015

[11] A. Yaacoub, "Flux de l'Information en Programmation Logique" Universite Paul Sabatier - Toulouse III, These de Doctorat, 2012.

[12] A. Yaacoub, A. Awada, and H. Kobeissi, "Information Flow in Concurrent Logic Programming." British Journal of Mathematics & Computer Science, Vol. 5, Issue.: 3, P. 367-382, 2015.

[13] A. Yaacoub, and A. Awada, "Information Flow in Datalog Using Very Naive and Semi Naive Bottom-Up Evaluation Techniques." International Journal of Computer Science: Theory and Application (Ijcsta), Vol. 5, Issue.: 2, P. 35-48, 2016.

[14] A. Yaacoub, "Bisimilarity, Datalog and Negation." International Journal of Computer Science: Theory and Application (Ijcsta), Vol. 5, Issue.: 2, P. 28-34, 2016.

[15] J.W. Lloyd, "Foundations of Logic Programming," 2nd Edition. Springer, 1987.

[16] C. Stefano, G. Georg, and T.Letizia, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)." Knowledge and Data Engineering, IEEE Transactions, Vol. 1, Issue.: 1, P. 146–166, 1989.

[17] J.Ullman, "Principles of Databases and Knowledge Base Systems," Volume I and II. Computer Science Press, 1988.

[18] S. Abiteboul, R. Hull, and V. Vianu, "Foundations of Databases." Addison-Wesley Reading, 1995.

[19] Y. Hinz, "Datalog Bottom-Up is the Trend in the Deductive Database Evaluation Strategy." Tech. Rep. INSS 690, University Of Maryland, 2002.

[20] R.Ramakrishnan, and J. Ullman, "A Survey of Research on Deductive Database Systems." Journal of Logic Programming, Vol. 23, P. 125–149, 1993.

[21] J. Ullman, "Bottom-Up Beats Top-Down for Datalog." Proceedings of The Eighth ACM Sigact-Sigmodsigart Symposium on Principles of Database Systems, P. 140–149, 1989.

[22] M. Vardi, "The Complexity of Relational Query Languages (Extended Abstract)." Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, P. 137–146, 1982