Inferring Secrets by Guided Experiments

Quoc Huy Do, Richard Bubel, and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
{do,bubel,haehnle}@cs.tu-darmstadt.de

Abstract. A program has secure information flow if it does not leak any secret information to publicly observable output. A large number of static and dynamic analyses have been devised to check programs for secure information flow. In this paper we present an algorithm that can carry out a systematic and efficient attack to automatically extract secrets from an insecure program. The algorithm combines static analysis and dynamic execution. The attacker strategy learns from past experiments and chooses as its next attack one that promises maximal knowledge gain about the secret. The idea is to provide the software developer with concrete information about the severity of an information leakage.

Keywords: information flow, symbolic execution, static analysis

1 Introduction

Information flow security is concerned with the development of methods that ensure that programs do not leak secret information, i.e., that it is not possible to learn secret information by looking at publicly accessible output.

To ensure that programs have secure information flow relative to a given information flow policy, a large number of static analyses have been devised (see [27] for a survey). Most of these approaches are *qualitative*, in the sense that they try to establish that a program is secure and they reject programs as insecure otherwise. In case of a leak (even if allowed by a given declassification policy) they do not provide details about how much information is leaked. *Quantitative* information flow analysis [1,2,3,29,17,25] complements qualitative analyses by measuring the amount of leaked information. Developers can use this feedback to decide whether the leakage is acceptable or not.

Our aim is to support detection and comprehension of information flow leaks during software development. In previous work [11] we presented an approach to generate demonstrator code for leakages in the form of failing tests. These tests could be examined and debugged by a developer to fix the leak. The generated tests merely demonstrated that a program does not respect a given information flow policy, but it was not possible to extract actual secrets. Extracting a secret or at least narrowing down the number of possible values of a secret information helps in two ways: (i) the software developer obtains additional information about the nature of the leak and (ii) it makes it easier to judge the severity of a leak and provides arguments to assign its fix an appropriate priority.

The work presented in this paper applies techniques developed for quantified information flow analysis to guide the systematic creation of an (as small as possible) set of experiments/attacks to be conducted to gain maximal knowledge about a secret. The set of experiments is built incrementally. New experiments are added only if they are non-redundant and lead to a "maximal" knowledge gain. This sets our approach apart from previous approaches [17,3,25] that use a random set of experiments (or simply state the existence of such a set) and enables us to obtain a tighter characterisation of secrets.

We introduce a novel approach for automatic generation of a "good" experiment set to exploit information flow leaks. The main contributions are: (i) an algorithm that combines static analysis and dynamic analysis. Symbolic execution is used to statically analyse a program's behaviour, to compute path conditions and symbolic states. Based on this information, knowledge about a secret is incrementally increased by devising knowledge-maximizing experiments that in turn refine the static analysis results. These experiments are obtained by (ii) maximizing information leakage relative to metrics that *depend on public input*. The result of our algorithm is a (iii) logical characterisation of a secret. Hence, a model finder can be used to extract the remaining candidates for the secret, and in the best case, the secret itself as the only remaining model.

The paper is structured as follows: In Section 2 we give the necessary background to make the paper self-contained. Section 3 is about our approach and its design. Section 4 describes the generation of the input values for the experiments with a focus on efficiency. An experimental evaluation is presented in Section 5. We finish with related work (Section 6) and conclusions/future work (Section 7).

2 Background

The programming language used throughout the paper is a simple, deterministic and imperative language with global variables of a 32-bit integer type (we denote their domain with $\mathbb{Z}_{32}$). For ease of presentation, we restrict ourselves to programs where termination is guaranteed for all inputs. Our implementation supports a rich subset of sequential Java, including method calls, objects with integer fields, and integer-typed arrays (see Section 5.1).

In the remaining paper we use p to denote a program and $\mathit{Var} = \{x_1, \ldots, x_n\}$ to denote an ordered set of all program variables occurring in p.

2.1 Symbolic Execution

Symbolic execution (SE) is a versatile static analysis technique [16]. Symbolic execution "runs" a program with symbolic (input) values instead of concrete ones.

Example 1. The program in Listing 1.1 uses `l`, `h` as program variables. For values of `l` below 100, the computed value stored in `l` represents the result of comparing the initial values of `l` and `h`, where `l` is assigned 3, 0, -3 for `l` being equal, less than, and greater than `h`, respectively. For values of `l` of 100 and above, the value 2 is assigned to `l`.

We start symbolic execution (SE) at the first statement in line 1 in an initial state where `l` and `h` have symbolic input values l_0 and h_0, respectively (short: $l : l_0$, $h : h_0$). This causes a split into two SE paths. The first branch deals with the case where the *branch condition* is $l_0 < 100$ and the second branch with the complementary case. We continue symbolic execution on the first branch with the **`if`**-statement in line 2. This causes another split with branch conditions $l_0 \doteq h_0$ and $l_0 \neq h_0$. Continuing again with the first branch, the next statement to be symbolically executed is the assignment of value 3 to `l` in line 3. We treat concrete values such as 3 as special symbolic values with a canoncial interpretation. □

Listing 1.1: Running Example

```
1 if (l < 100) {
2   if (l == h)
3     l = 3;
4   else
5     if (l < h) l = 0; else l = -3;
6   }
7 else
8   l = 2;
```

Symbolic execution creates an SE tree representing all possible concrete execution paths; moreover, a *single* symbolic execution path may represent infinitely many concrete execution paths. Each node of an SE path corresponds to a code location and contains the symbolic state at that point: a mapping from program variables to their symbolic value and a path condition. The *path condition* is obtained as the conjunction of all branch conditions up to the current point of execution and unambiguously defines the execution path to be taken. The initial state of any execution path through a node with path condition *pc* must necessarily satisfy *pc*.

Path conditions and symbolic values are always expressed relative to the initial symbolic values present in the initial symbolic state. For instance, assume we symbolically execute the program fragment

```
l=l+1; h=h*2; if (l>h){l=5;}
```

in initial state $(l : l_0, h : h_0)$ and initial path condition `true`. Then the resulting SE tree has two branches, one with the final symbolic state $(l : 5, h : 2 * h_0)$ and path condition $l_0 + 1 > 2 * h_0$ and the other branch with the final symbolic state $(l : l_0 + 1, h : 2 * h_0)$ and path condition $l_0 + 1 \leq 2 * h_0$. In the following, instead of introducing a new constant symbol v_0 to refer to the initial value of a program variable v, we simply use the program variable v itself. This means program variables occurring in path conditions and symbolic values refer always to their initial value.

We use SET_p to refer to the SE tree of program p and N_p to refer to the number of symbolic execution paths of SET_p. For each leaf node of an SE path i $(1 \leq i \leq N_p)$ the corresponding path condition is denoted with pc_i and the symbolic value of variable $v \in \mathit{Var}$ in the final state of path i is denoted with the expression f_i^v. Later we need to express symbolic values or path conditions over a different variable signature: Let $V = \{x_1, \ldots, x_n\}$, $V' = \{x'_1, \ldots, x'_n\}$ be ordered, disjoint sets of program variables with the same cardinality; we write $pc_i[V'/V]$, meaning that each x_i in pc_i has been replaced by x'_i. In case of two disjoint variables sets V_1, V_2 we write $pc_i[V'_1, V'_2 \;/\; V_1, V_2]$ instead of $pc_i[V'_1/V_1][V'_2/V_2]$. Similar for the symbolic values f_i^v.

There are several approaches to deal with loops and recursive method calls in SE to achieve a finite SE tree. We follow the approach presented in [14], which uses specifications, namely, method contracts and loop invariants. In case of sound and complete specifications this approach is fully precise. In case of incomplete specifications, completeness (but not soundness) is sacrificed. In brief, the effect of loops and method calls is encoded as part of the path condition and the introduction of fresh symbolic values.

The approach presented in this paper extends our previous work [11] in which SE is used to compute path conditions and the final symbolic values of program variables to obtain a logic characterisation of insecurity. We recapture the most important ideas: Let L, H be a partitioning of Var. The noninterference policy $H \not\rightsquigarrow L$ forbids any information flow from the initial value of high (confidential/secret) program variables H to low (public) variables L. In [9,10] self-composition is introduced as a means to formalise in terms of a logic formula whether or not a program is secure realtive to a given noninterference policy. The negation of that formula is true for insecure programs, i.e. any model of the negated formula describes a pair of program runs that leak information. We used that idea as follows: Given two SE paths i and j with path conditions pc_i, pc_j and final symbolic values f_i^v, f_j^v, $v \in \mathit{Var}$. The *insecurity* formula

$$\mathit{Leak}(i,j) \equiv (\bigwedge_{v \in L} v \doteq v') \wedge pc_i \wedge (pc_j[\mathit{Var}'/\mathit{Var}]) \wedge \bigvee_{v \in L} f_i^v \not\doteq (f_j^{v'}[\mathit{Var}'/\mathit{Var}]) \qquad (1)$$

has a model (an assignment of values to program variables satisfying (1)) if there are two program runs, one taking path i and the other one path j $(i = j$ possible), that end in final states differing in the value of at least one low variable, even though their initial states coincided on the low input. Our programs are deterministic, hence, this can only be the case if the value of high variables influenced the final value of the low variables. To check whether a program is insecure, we compare all pairs of symbolic execution paths:

$$\bigvee_{1 \leq i \leq j \leq N_p} \mathit{Leak}(i,j) \qquad (2)$$

An SE path that leaks information is called a *risky path*. The set of all risky paths is denoted by *Risk*. Details on how to support other information flow policies than noninterference can be found in [11].

2.2 Quantitative Information Flow Analysis

We recall some measures for quantifying information leaks [31,28,18,3]. Given a program p and a noninterference policy $H \not\rightsquigarrow L$. Let $O \subseteq L$ (usually: $O = L$) be a subset of low variables whose value can be observed by an attacker after termination of p. We assume that before running p, the attacker knows about the values of low variables (or can even manipulate them); and that the initial values of variables in H and L are independent (i.e. from an attacker's perspective knowledge about L does not entail any knowledge about H).

Information leakage from H to O can be seen as the reduction of uncertainty of the attacker about the values of H that can be achieved by observing the final values of the variables in O after a run of program p:

$$\text{information leaked} = \text{initial uncertainty} - \text{remaining uncertainty}$$

To measure uncertainty a number of entropies can be used, for instance, Shannon entropy [6,26], min entropy [28] or guessing entropy [18,3]. To quantify information leakage, we use the approach described in [31].

Let $\mathbb{L}, \mathbb{H}$ denote the finite sets of all possible values of L and H, e.g., for two unrestricted integer program variables $H = \{h_1, h_2\}$, $\mathbb{H}$ is the Cartesian product $\mathbb{Z}_{32} \times \mathbb{Z}_{32}$ of their domain. Similarly, let $\mathbb{O}$ be the set of all possible output values of O. Let the function $\mathbb{O}_D : \mathbb{L} \to 2^{\mathbb{O}}$ that computes the set of all possible output values of O for a given low input be defined as follows:

$$\mathbb{O}_D : \bar{l} \mapsto \{\bar{o} \mid \bar{o} \text{ final values of } O \text{ after executing } p(\bar{l}, \overline{h}), \text{for each } \overline{h} \in \mathbb{H}\}$$

Each low input value $\bar{l}$ defines a random variable $O_{out}(\bar{l})$ corresponding to the observed output values in the set $\mathbb{O}_D(\bar{l})$ after running program p with fixed low level input $\bar{l}$. We denote with $O_{out}(L)$ the function from $\mathbb{L}$ to the space of random variables as defined above. The random variables corresponding to the initial values of H are denoted with H_{in}.

The following subsections describe different measures to compute information leakage of a program p with parameter L. We discuss three possible definitions of leakage: $\mathtt{ShEL_p}(L)$, $\mathtt{MEL_p}(L)$ and $\mathtt{GEL_p}(L)$ based on Shannon entropy, min entropy and guessing entropy, respectively. All logarithms are base 2.

Shannon entropy-based leakage

Definition 1. *Given random variables X, Y with sample space $\mathbb{X}$ and $\mathbb{Y}$, respectively. The* Shannon entropy *of X is defined as*

$$\mathcal{H}(X) = -\sum_{x \in \mathbb{X}} P(X = x) log(P(X = x))$$

and the conditional Shannon entropy *of X given Y as*

$$\mathcal{H}(X|Y) = \sum_{y \in \mathbb{Y}} P(Y = y) \mathcal{H}(X|Y = y)$$

where $\mathcal{H}(X|Y = y) = -\sum_{x \in \mathbb{X}} P(X = x|Y = y) log(P(X = x|Y = y))$.

Intuitively, $\mathcal{H}(X)$ is the average number of bits required to encode the values of X and $\mathcal{H}(X|Y=y)$ quantifies the average number of bits needed to describe the outcome of X under the condition that the value of Y is known.

Shannon entropy and its conditional variant are used to quantify information leakage as follows: the initial uncertainty of the attacker about H_{in} is interpreted as Shannon entropy of H_{in}, while the remaining uncertainty of the attacker about H_{in} when $O_{out}(L)$ is known is interpreted as conditional entropy. Then the information leakage is the *mutual information* of H_{in} and $O_{out}(L)$:

$$\mathtt{ShEL_p}(L) = I(H_{in}; O_{out}(L)) = \mathcal{H}(H_{in}) - \mathcal{H}(H_{in}|O_{out}(L))$$

Because mutual information is symmetric, we get

$$\mathtt{ShEL_p}(L) = I(H_{in}; O_{out}(L)) = \mathcal{H}(O_{out}(L)) - \mathcal{H}(O_{out}(L)|H_{in})$$

As our programs are deterministic, only one value of $O_{out}(L)$ corresponds to a value of H_{in}. It is easy to see that $\mathcal{H}(O_{out}(L)|H_{in}) = 0$. Finally, we end up with

$$\mathtt{ShEL_p}(L) = \mathcal{H}(O_{out}(L)) \tag{3}$$

Min entropy-based leakage While Shannon entropy is a natural way to quantify leakage, it fails to reflect the vulnerability that high values might be guessed correctly in a single try. Consider the two programs

$p_1 \equiv$ `if (h%8==0)l=h else l=1),` $\quad p_2 \equiv$ `l=h&0777`

taken from [28]. Using Shannon entropy, the mutual information leakage of program p_1 is smaller than that of p_2, i.e., p_1 is considered to be more secure than p_2. However, the risk of leaking the complete value of `H` in a single run is significantly higher for p_1 than for p_2. Smith [29] proposed *min entropy* as an alternative metric to address this problem. As Smith focuses on programs without low input, we use the extension given in [31]:

Definition 2. *Given random variables X, Y with sample space $\mathbb{X}$ and $\mathbb{Y}$, respectively. The* min entropy *of X is defined as $\mathcal{H}_\infty(X) = -log\mathcal{V}(X)$ and the* conditional min entropy *of X given Y as $\mathcal{H}_\infty(X|Y) = -log\mathcal{V}(X|Y)$ where $\mathcal{V}(X) = max_{x\in\mathbb{X}}P(X=x)$ and $\mathcal{V}(X|Y) = \sum_{y\in\mathbb{Y}} P(Y=y)max_{x\in\mathbb{X}}P(X=x|Y=y)$.*

Intuitively, the min entropy of X represents the highest probability that X can be guessed in a single try. Using min entropy allows to measure information leakage as follows: the initial uncertainty is interpreted as min entropy of H_{in} and the remaining uncertainty is the conditional min entropy of H_{in} given O_{out}. The min-entropy-based leakage becomes then

$$\mathtt{MEL_p}(L) = \mathcal{H}_\infty(H_{in}) - \mathcal{H}_\infty(H_{in}|O_{out}(L)) = \log\frac{\mathcal{V}(H_{in}|O_{out}(L))}{\mathcal{V}(H_{in})}$$

Smith [29] shows for deterministic programs under the assumption that H_{in} is uniformly distributed that for a given program p

$$\mathtt{MEL_p}(L) = \log|\mathbb{O}_D(L)| \tag{4}$$

Guessing entropy-based leakage

Definition 3. *Given random variables X, Y with sample space $\mathbb{X}$ and $\mathbb{Y}$, respectively. The* guessing entropy *of X is defined as*

$$\mathcal{G}(X) = \sum_{1 \le i \le m} i \cdot P(X = x_i) \qquad (m = |\mathbb{X}|)$$

where $x_1, \dots, x_m$ satisfy $\forall i, j.(i \le j \rightarrow P(X = x_i) \ge P(X = x_j))$. The conditional guessing entropy of X given Y is defined as

$$\mathcal{G}(X|Y) = \sum_{y \in \mathbb{Y}} P(Y = y)\mathcal{G}(X|Y = y)$$

where

$$\mathcal{G}(X|Y = y) = \sum_{1 \le i \le m} i \cdot P(X = x|\,Y = y)$$

and $x_1, \dots, x_m$ satisfy $\forall i, j.(i \le j \rightarrow P(X = x_i|\,Y = y) \ge P(X = x_j|\,Y = y))$.

Intuitively, the *guessing entropy* of a random variable X is the average number of questions of the kind: "Is the value of X equal to x?" that are needed to infer the value of X correctly [20].

The derivation of the computation of the guessing entropy-based leakage is similar to the previous ones yields:

$$\texttt{GEL}_{\texttt{p}}(L) = \mathcal{G}(H_{in}) - \mathcal{G}(H_{in}|O_{out}(L)) \tag{5}$$

3 Automatic Inference of a Program's Secrets

This section describes our attacker model and presents the core logic of our algorithm to automatically infer a program's secrets.

3.1 Attacker Model and Overview

We assume that the attacker knows the source code and can run the program multiple times to observe public outputs. The notation p, L, H, etc. is as above.

Fig. 1 shows an overview of our approach. First, the source code is analysed statically by symbolic execution to identify execution paths, called risky paths, that might cause information leakage (directly or indirectly). Based on this analysis a number of experiments is performed to infer the secret. An experiment is a program run with concrete input together with the outcome. To perform an experiment the algorithm selects suitable low input based on knowledge about risky execution paths and knowledge accumulated in previous runs. The algorithm terminates when one of the following conditions holds: (i) all secrets have been inferred unambiguously; (ii) it can be determined that no new knowledge can be inferred; (iii) a specified limit of concrete program runs is reached.

The algorithm assumes that high variables are not modified by or in between concrete runs. We use $\overline{h}_s \in \mathbb{H}$ to refer to a secret, i.e.. concrete (to us unknown) values of H.

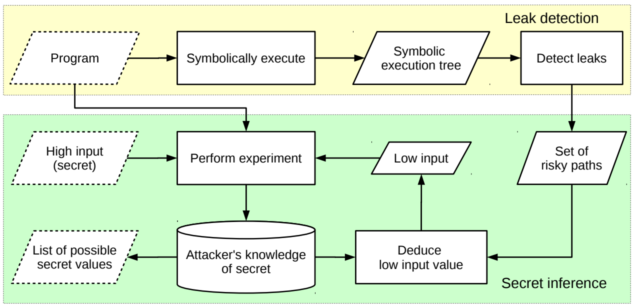

Fig. 1: Structure of the algorithm to infer secrets

3.2 Knowledge Representation of High Input

We fix a program p, a noninterference policy $H \not\rightsquigarrow L$, and a set $O \subseteq L$ of observable low variables. The concrete value sets $\mathbb{L}, \mathbb{H}, \mathbb{O}_D(\cdot)$ are as before. To gain knowledge about a secret, a series of experiments is performed.

Definition 4. *A pair $\langle \bar{l}, \bar{o} \rangle$ with $\bar{l} \in \mathbb{L}$, $\bar{o} \in \mathbb{O}_D(\bar{l})$ is called an* experiment *for p. By convention, we denote with $\bar{h}_s$ the high input value that was used in the run.*

Let $E = \{\langle \bar{l}_j, \bar{o}_j \rangle \mid 1 \leq j \leq m\}$ be a set of experiments for a program p. Symbolic execution of p yields a precise logical description of all reachable final states, see Section 2. Recall that N_p is the number of all feasible symbolic execution paths. For each symbolic execution path i, we obtain its path condition pc_i and the final symbolic values f_i^v of any program variable v. Let O' be an ordered set of fresh program variables such that for any $v \in O$ there is a corresponding $v' \in O'$ and the cardinality of O and O' is equal, i.e. $|O| = |O'|$. The formula

$$Info(L, H, O') = \bigvee_{1 \leq i \leq N_p} InfoPath_i(L, H, O') \tag{6}$$

where $InfoPath_i(L, H, O') = pc_i \wedge \bigwedge_{v' \in O'} v' = f_i^v$ "records" the information about final values contained in a symbolic execution path. It is true whenever the variables in H, L, O' are assigned values $\bar{h}$, $\bar{l}$, $\bar{o}$ such that executing p in an initial state $\langle \bar{l}, \bar{o} \rangle$ terminates in a final state where the variables in O have values $\bar{o}$. For a concrete experiment $\langle \bar{l}, \bar{o} \rangle$ formula (6) is instantiated to

$$Info_{\langle \bar{l}, \bar{o} \rangle}(H) = Info(\bar{l}, H, \bar{o}) = Info(L, H, O')[\bar{l}, \bar{o} \ / \ L, O'] \tag{7}$$

This formula must true at the time of running the experiment, because (i) the taken execution path must be contained in one of the symbolic execution paths, and (ii) the observed output values must be equal to those obtained by evaluating the symbolic values with the concrete initial values of the low and high variables.

We write $Info_{\langle \bar{l}, \bar{o} \rangle}(H)$ to emphasize that the truth value of the formula only depends on the assignment of concrete values to the program variables in H. The formula $\mathit{Info}_{\langle \bar{l},\bar{o}\rangle}(H)$ constrains the possible high values and can be seen as the information about $\overline{h}_s$ that can be learned from experiment $\langle \bar{l}, \bar{o}\rangle$. The *knowledge* about $\overline{h}_s$ gained from all experiments in a set E is then

$$K^E(H) = K^{\emptyset}(H) \wedge \bigwedge_{\langle \bar{l},\bar{o}\rangle \in E} \mathit{Info}_{\langle \bar{l},\bar{o}\rangle}(H) \tag{8}$$

where $K^{\emptyset}(H)$ is the *initial knowledge* about $\overline{h}_s$ that is known before performing any experiment, for example, domain restrictions. If nothing is known about $\overline{h}_s$, then the initial knowledge $K^{\emptyset}(H)$ is simply *true*. The set of all models of $K^E(H)$ contains by construction also the actual secret $\overline{h}_s$ (a simple inductive argument with base case that $K^{\emptyset}(H)$ is satisfied by $\overline{h}_s$).

We want to design a set of experiments that reduces, as much as possible, the number of possible concrete values for H that satisfy (8). The smaller this number is, the more we succeeded to narrow down the possible values for the secret. In particular, if only one possible value remains, we know the secret.

The set of all values of a variable set X that satisfy a formula $\varphi(X)$ is denoted by $\mathit{Sat}(\varphi)$. Hence, $\mathit{Sat}(K^E(H))$ is the set of all values of H that satisfy $K^E(H)$. As usual we use $|S|$ to denote the cardinality of a set S.

Example 2. Consider again the program from Listing 1.1 with `l` as low variable and `h` as high variable. Assume that the value of `h` is 10. Initially, the knowledge about the value of `h` is its domain $-2^{31} \leq \mathtt{h} < 2^{31}$.

Given two experiment sets $X = \{\langle 5, 0\rangle, \langle 3, 0\rangle, \langle 8, 0\rangle\}$, $Y = \{\langle 5, 0\rangle, \langle 17, -1\rangle\}$. The knowledge about the secret input value of `h` that can be gained from X and Y is $K^X(\{\mathtt{h}\}) = 8 < \mathtt{h} < 2^{31}$ and $K^Y(\{\mathtt{h}\}) = 5 < \mathtt{h} < 17$, respectively. Even though $|X| > |Y|$, it is obvious that $|\mathit{Sat}(K^Y(\{\mathtt{h}\}))| \ll |\mathit{Sat}(K^X(\{\mathtt{h}\}))|$, hence the knowledge about the secret value of `h` obtained from Y is higher than the one obtained from X. □

We want to accumulate maximal knowledge about a secret with as few experiments as possible. In particular, we do not want to perform experiments that do not create any knowledge gain. Avoiding redundant experiments is essential to achieve performance.

Definition 5. *An experiment $\langle \bar{l}, \bar{o}\rangle$ is called* redundant *for $K^E(H)$ if the following holds:*

$$\forall \overline{h}.(K^E(\overline{h}) \rightarrow \mathit{Info}_{\langle \bar{l},\bar{o}\rangle}(\overline{h}))$$

A redundant experiment $\langle \bar{l}, \bar{o}\rangle$ gains no new information about a secret $\overline{h}_s$ for knowledge $K^E(H)$, because $K^E(\overline{h}) \wedge \mathit{Info}_{\langle \bar{l},\bar{o}\rangle}(\overline{h}) \equiv K^E(\overline{h})$.

3.3 Algorithm for Inferring High Input

Algorithm 1 implements the core of our approach. The result is a logical formula that represents the accumulated knowledge about the high variables the algorithm was able to infer. The result can be used as input to an SMT solver or another model finder to compute concrete models representing possible secrets.

```
Data: $p$: program to be attacked (with the high input already set);
      noninterference policy $H \not\rightsquigarrow L$; $O \subseteq L$: observable low variables; $K^{\emptyset}(H)$:
      initial knowledge about $H$; $maxE$: maximum number of experiments
Result: $K^E(H)$: the accumulated knowledge about $H$ obtained by executing
        the experiments $E$
$E \leftarrow \emptyset$;
$K \leftarrow K^{\emptyset}(H)$;
while $|E| < maxE$ do
    $(\bar{l}, leakage) \leftarrow findLowInput(E, K)$;
    if $leakage > 0$ then
        execute $p$ with low input $\bar{l}$;
        $\bar{o} \leftarrow$ values of $O$ when $p$ terminates;
        $E \leftarrow E \cup \langle \bar{l}, \bar{o} \rangle$;
        $K \leftarrow K \wedge Info_{\langle \bar{l}, \bar{o} \rangle}(H)$;
        if $|Sat_H(K)| = 1$ then
            exit while;
        end
    else
        exit while;
    end
end
```

Algorithm 1: Secret inference

Algorithm 1 receives as input the program p, the symbolic execution result for p, i.e. p's SE tree together with all path conditions and symbolic values in the final symbolic execution state, the attacker's initial knowledge, etc. In particular, the formula $Info_{\langle \bar{l}, \bar{o} \rangle}(H)$ can be computed.

First, the set of already performed experiments E is initialized with the empty set and the accumulated knowledge K is initialized with the initial knowledge of the attacker. Thereafter, the main loop of the algorithm is entered. At the beginning of each iteration K contains the accumulated knowledge of all experiments executed up to now, i.e. $K = K^E(H)$. At the beginning of each loop iteration the low input $\bar{l}$ for a new experiment is determined by method *findLowInput*(E, K) based on the set of experiments E and the knowledge K accumulated so far. That method returns also a measure of the leakage expected to be observed by executing p with the provided low input. The method returns 0 as leakage only if all low input values would result in redundant experiments. In its most rudimentary implementation the method returns simply random values and a positive number for the leakage. We discuss more refined implementations in Section 4.

In case the expected leakage is positive (i.e. something new might be learned), program p is executed with the computed low input $\bar{l}$ and the set of experiments is extended by the pair $\langle \bar{l}, \bar{o} \rangle$ where $\bar{o}$ are the values of the observable variables when p terminates. In the next step we update the accumulated knowledge by adding the conjunct $Info_{\langle \bar{l}, \bar{o} \rangle}(H)$. Afterwards, we check whether the accumulated knowledge uniquely determines the values of the high variables. If this is the case we know the exact secret and return. Otherwise, we continue another loop iteration until the maximal number of experiments *maxE* is reached. In case that the expected leakage is zero, no useful low input can be found any longer and the algorithm terminates.

4 Finding Optimal Low Input

We discuss method *findLowInput*(E) in detail and aim to provide more efficient implementations than the trivial one sketched above. The main purpose of the method is to determine optimal low input values that lead to a maximal gain of knowledge about the values of the high variables. We use the security metrics discussed in Sect. 2 to guide this process and show how these can be effectively computed by employing symbolic execution and parametric model counting.

4.1 Risky Paths and Reachable Paths

We start with a set of experiments E ($|E| = m$) and the accumulated knowledge about the high variables in form of the logic formula $K^E(H)$. We assume the initial knowledge about secret $K^{\emptyset}(H)$ is correct ($\overline{h}_s$ satisfies $K^{\emptyset}(H)$), hence $\overline{h}_s$ also satisfies $K^E(H)$. Our aim is to find the low level input $\overline{l}_{m+1}$ for a new experiment that is most promising for a maximal knowledge gain. In this subsection we discuss how to avoid generation of low input that would lead to a redundant experiment.

A *risky path* is a symbolic execution path which might contribute to an information leakage (see Section 2.1).

Definition 6. *Let p be a program and N_p be the number of all symbolic paths of p. A symbolic path i ($1 \leq i \leq N_p$) is called a* risky path *for a noninterference policy $H \not\rightsquigarrow O$ iff $\exists k.(1 \leq k \leq N_p \wedge Leak(i,k))$ is satisfiable. The set of all risky paths of p is denoted with $Risk$.*

Definition 7. *An input value $\overline{l}$ of L is called* i-matched *for $K^E(H)$ iff the formula $\forall \overline{h}.(K^E(\overline{h}) \rightarrow pc_i[\overline{l}, \overline{h} \ / \ L, H])$ holds.*

Intuitively, if a concrete low input $\overline{l}$ is i-matched for $K^E(H)$ and $K^E(H)$ is correct, then $pc_i[\overline{l}/L]$ holds for all possible values of high inputs.

Since all path conditions are mutually exclusive, we can conclude that if $\overline{l}$ is i-matched for $K^E(H)$ and $\langle \overline{l}, \overline{o} \rangle$ is the corresponding experiment, then $K^E(H) \wedge Info_{\langle \overline{l}, \overline{o} \rangle}(H) \equiv K^E(H) \wedge InfoPath_i(\overline{l}, H, \overline{o})$. If $K^E(H) = true$ then we have $Info_{\langle \overline{l}, \overline{o} \rangle}(H) \equiv InfoPath_i(\overline{l}, H, \overline{o})$.

Lemma 1. *If $\overline{l}$ is i-matched for $K^E(H)$ for some $i \notin Risk$, then any program run with input $\overline{l}$ leads to the same output $\overline{o} \in \mathbb{O}_D(\overline{l})$ for all high inputs $\overline{h}$ for which $K^E(\overline{h})$ holds.*

Theorem 1. *If $\overline{l}$ is i-matched for $K^E(H)$ and some $i \notin Risk$, then experiment $\langle \overline{l}, \overline{o} \rangle$ is redundant.*

The proofs of Lemma 1 and Theorem 1 are in Appendices A.1 and A.2, respectively. Theorem 1 has the following corollary:

Corollary 1. *$InRisk(L)$ denotes the formula $\exists\overline{h}.\big(K^E(\overline{h}) \wedge \bigwedge_{i\notin Risk} \neg pc_i[\overline{h}\,/\,H]\big)$. If for some $\bar{l} \in \mathbb{L}$ the formula $InRisk(\bar{l})$ is false then the experiment $\langle\bar{l},\bar{o}\rangle$ is redundant for $K^E(H)$.*

Example 3. The SE tree of the program in Listing 1.1 has four paths with path conditions $pc_1 = \mathtt{l} < 100 \wedge \mathtt{l} = \mathtt{h}$, $pc_2 = \mathtt{l} < 100 \wedge \mathtt{l} < \mathtt{h}$, $pc_3 = \mathtt{l} < 100 \wedge \mathtt{l} > \mathtt{h}$ and $pc_4 = \mathtt{l} \geq 100$. The set of risky paths is $Risk = \{1, 2, 3\}$. The fourth path is not a risky path as it does not contribute to any leak. We have $InRisk(\{\mathtt{l}\}) = \exists h.\neg(\mathtt{l} \geq 100) \equiv \mathtt{l} < 100$ indicating that only low input values less than 100 may lead to any information gain. □

Definition 8. *An SE path i is called a* reachable path *for $K^E(H)$ iff the following formula is satisfiable:*

$$K^E(H) \wedge pc_i \tag{9}$$

R^E denotes the set of all reachable paths for $K^E(H)$.

Example 4. (Example 3 cont'd) Assume the initial knowledge about the value of $\mathtt{h}$ is $-2^{31} \leq \mathtt{h} < 2^{31}$ and the secret value of $\mathtt{h}$ is 1000. We execute the program in Listing 1.1 with $\mathtt{l} = 98$. The execution terminates in a state where $\mathtt{l}$ has been set to 0. Using this experiment, we obtain as accumulated knowledge about $\mathtt{h}$: $-2^{31} \leq \mathtt{h} < 2^{31} \wedge ((98 = \mathtt{h} \wedge 3 = 0) \vee (98 < \mathtt{h} \wedge 0 = 0) \vee (98 > \mathtt{h} \wedge -3 = 0)) \equiv 98 < \mathtt{h} < 2^{31}$. With this knowledge about $\mathtt{h}$, the risky path 3 becomes unreachable because the formula $98 < \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{l} > \mathtt{h}$ is unsatisfiable. □

Theorem 2. *For all experiments $\langle\bar{l},\bar{o}\rangle$, it holds that*

$$K^E(H) \wedge Info_{\langle\bar{l},\bar{o}\rangle}(H) \equiv K^E(H) \wedge \bigvee_{i\in R^E} InfoPath_i(\bar{l}, H, \bar{o})$$

Theorem 2 (Proof in Appendix A.3) shows that all unreachable paths can be ignored while constructing the knowledge about $\overline{h}_s$. Moreover, it allows us to consider only reachable paths when deducing optimal low input, which we explain in the next sections.

4.2 Quantifying Leakage by SE

We denote the number of assignments of values to the variables in H that satisfy $K^E(H)$ by $S_E = |Sat(K^E(H))|$. We assume further that the probability distribution of the values for H is uniform and the actual value of H satisfies $K^E(H)$. ($K^E(H)$ is correct).

Definition 9. *For a formula g, let V be the set of all program variables occurring in g and let $V = X \,\dot{\cup}\, Y$ be a partitioning. Function $\mathtt{C}_X[Y](g)$ is called* parametric counting function *iff it returns the number of assignments to the variables of X that satisfy g (i.e. the number of models) as a function of Y.*

Example 5. Given $V = \{\mathtt{h}, \mathtt{l}\}$ and $g = 0 \leq \mathtt{h} < 100 \wedge \mathtt{h} \geq \mathtt{l} \wedge 0 \leq \mathtt{l} < 100$. Then the number of models of $\mathtt{h}$ satisfying g depends on $\mathtt{l}$ and can be determined for any value of $\mathtt{l}$ satisfying $0 \leq \mathtt{l} < 100$ by $\mathtt{C}_{\{\mathtt{h}\}}[\{\mathtt{l}\}](g) = 100 - \mathtt{l}$. □

We want to extend the experiment set E by adding a new experiment $\langle \bar{l}, \bar{o} \rangle$ such that the observable leakage (knowledge gain on high variables) is as high as possible. The following theorem (Proof in Appendix A.4) provides an iterative method to compute the different leakage measures from Section 2.2 based on counting the models of $K^E(H)$.

Theorem 3. *Let E be an experiment set and $K^E(H)$ the knowledge about the high variables. Then the Shannon entropy-based $\mathtt{ShEL}_\mathtt{p}(L)$, the Min entropy-based $\mathtt{MEL}_\mathtt{p}(L)$, and the Guessing entropy-based $\mathtt{GEL}_\mathtt{p}(L)$ leakages can be computed as follows:*

$$\mathtt{ShEL}_\mathtt{p}(L) = \mathit{log}(S_E) - \frac{1}{S_E} \sum_{\bar{o} \in \mathbb{O}_D(L)} \big(\mathtt{C}_H[L](g(L, H, \bar{o}))\mathit{log}(\mathtt{C}_H[L](g(L, H, \bar{o}))) \big)$$

$$\mathtt{GEL}_\mathtt{p}(L) = \frac{S_E + 1}{2} - \frac{1}{2S_E} \sum_{\bar{o} \in \mathbb{O}_D(L)} \big(\mathtt{C}_H[L](g(L, H, \bar{o}))(\mathtt{C}_H[L](g(L, H, \bar{o})) + 1) \big)$$

$$\mathtt{MEL}_\mathtt{p}(L) = \mathit{log}(\mathtt{C}_{O'}[L](\exists \bar{h}. g(L, \bar{h}, O'))) \qquad (O' \textit{ as defined in Section 3.2})$$

where $g(L, H, O) = K^E(H) \wedge \mathit{InRisk}(L) \wedge \bigvee_{i \in R^E} \mathit{InfoPath}_i(L, H, O)$.

Under the assumption that pc_i and the symbolic observable output values $\overline{f}_i^O$ are linear, the computation of $\mathtt{C}_H[L](\ldots)$ and $\mathtt{C}_{O'}[L](\ldots)$ can be reduced to counting the number of integer points in parametric and non-parametric polytopes for which efficient approaches (and tools) exist [30].

4.3 Method *findLowInput*

Algorithm 2 shows detailed pseudo-code of method *findLowInput*. It computes the optimal low input values for a given leakage metric together with the computed leakage. First, the set of reachable paths R^E is determined by checking the reachability of all paths using formula (9). If no reachable paths exist or all reachable paths are not risky, the algorithm exits and returns 0 as leakage value (in that case the low input values are irrelevant). Otherwise, the optimal low input values for the leakage metric are computed.

Here $QLeak(L)$ is one of $\mathtt{ShEL}_\mathtt{p}(L)$, $\mathtt{GEL}_\mathtt{p}(L)$, $\mathtt{MEL}_\mathtt{p}(L)$ according to the chosen security metric. The low input values are determined by solving the optimization problem: $\mathit{argmax}_{\bar{l} \in \mathbb{L}} QLeak(\bar{l})$. In case of $\mathtt{ShEL}_\mathtt{p}(L)$ and $\mathtt{GEL}_\mathtt{p}(L)$ this is equivalent to minimizing the sum expression in their corresponding formula given in Theorem 3.

```
Data: Set of performed experiments $E$, current knowledge $K^E(H)$
Result: $(\bar{l}, leakage)$: optimal low input value and corresponding leakage
$R^E \leftarrow findAllReachablePaths(K^E(H))$;
if $|R^E| > 0 \wedge R^E \cap Risk \neq \emptyset$ then
    $QLeak(L) \leftarrow$ appropriatly instantiated entropy formula;
    $\bar{l} \leftarrow findL2Maximize(QLeak(L))$;
    if $\bar{l} = null$ then
        $\bar{l} \leftarrow$ random value that does not appear in $E$;
    end
    $leakage \leftarrow QLeak(\bar{l})$;
else
    $\bar{l} \leftarrow null$;
    $leakage \leftarrow 0$;
end
```

Algorithm 2: Implementation of method *findLowInput*

4.4 Choosing a Suitable Security Metric

Choosing the right security metric for a given program plays an important role for finding optimal low input values. The choice influences the computational complexity of the optimization problem as well as the quality of the found low input. It turns out that the Shannon entropy-based metric and the guessing entropy-based metric are significantly more expensive compared to the min entropy-based metric. The reason is that min entropy-based leakage merely requires to estimate the *cardinality* of the observable output values, while the two others require to *enumerate* each possible output value. On the other hand, they are able to find better low level input.

Consequently, the Shannon and guessing entropy-based leakage metrics are only feasible for programs whose observable output (i) either depends only on the chosen SE path, but not on the actual values of the low or high variables (i.e. each SE path assigns only constant values to the observable variables); (ii) or the output values depend only on the low input (i.e. for a specific concrete low input, their concrete value can be determined by evaluating the corresponding symbolic value f). For all other programs, determining the possible concrete output values is too expensive in practice. Our approach automatically selects the best possible metric. This works, because the class into which a program falls can be determined by reasoning about the symbolic output values. We illustrate (for space reasons only for case (i) described above) how the Shannon and guessing entropy-based leakage metrics can be used.

Let i be a reachable path with path condition pc_i and symbolic output values $\overline{f_i^O}$. By assumption (i), the symbolic values in $\overline{f_i^O}$ are constants (i.e. independent of any program variables), so they can be evaluated to concrete values $\overline{o}_i$. We may assume that the output values for all SE paths $i \neq j$ differ, hence $\overline{o}_i \neq \overline{o}_j$ (otherwise, paths i, j are merged into one with path condition $pc_i \vee pc_j$). Further, $\mathbb{O}_D(L) = \{\overline{o}_i | i \in R^E\}$, because we only consider reachable paths. Taking both observations together, we conclude that for all $i, j \in R^E$ with $i \neq j$ the formula $\mathit{InfoPath}_i(L, H, \overline{o}_j)$ is equivalent to false and $\mathit{InfoPath}_i(L, H, \overline{o}_i)$ simplifies to pc_i. We use this to simplify the definition of g in Theorem 3:

$$g(L, H, \overline{o}_i) \equiv K^E(H) \wedge pc_i$$

Now computation of $\mathtt{ShEL}_\mathtt{p}(L)$ and $\mathtt{GEL}_\mathtt{p}(L)$ becomes significantly cheaper, because the cardinality of the set of possible observable outputs is bounded by the number of reachable paths and only path conditions need to be taken into account.

Example 6. (Example 3 Cont'd) For our running example we already identified the set of risky paths as $\mathit{Risk} = \{1, 2, 3\}$ and obtained $\mathit{InRisk}(\mathtt{l}) = \mathtt{l} < 100$. A closer inspection of the program reveals the following: as long as our only knowledge about $\mathtt{h}$ is that its value is within an interval $[a, b]$ then choosing the arithmetic middle $\frac{b+a}{2}$ for the input value of $\mathtt{l}$ is the best choice.

The initial knowledge about $\mathtt{h}$ is that its value is between -2^{31} and $2^{31} - 1$, hence, the best choice is 0 or -1. We show that the solution computed *automatically* by our algorithm comes to the same conclusion. To avoid redundant experiments, we know already that $\mathtt{l}$ must be chosen such that $\mathtt{l} < 100$ $(= \mathit{InRisk}(\mathtt{l}))$. From the symbolic output values, we obtain $\mathbb{O}_{\{\mathtt{l}\}} \subseteq \{3, 0, -3\}$ and:

$$\begin{aligned}
\mathtt{g}(\mathtt{l}, \mathtt{h}, 3) &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} = \mathtt{l} \\
\mathtt{g}(\mathtt{l}, \mathtt{h}, 0) &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} > \mathtt{l} \\
\mathtt{g}(\mathtt{l}, \mathtt{h}, -3) &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} < \mathtt{l} \\
\mathtt{g}(\mathtt{l}, \mathtt{h}, \mathtt{l'}) &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \\
&\quad \big((\mathtt{l} = \mathtt{h} \wedge \mathtt{l}' = 3) \vee (\mathtt{l} < \mathtt{h} \wedge \mathtt{l}' = 0) \vee (\mathtt{l} > \mathtt{h} \wedge \mathtt{l}' = -3)\big)
\end{aligned}$$

where $\mathtt{l}'$ is a new program variable representing the final value of $\mathtt{l}$. Model counting (we used the tool Barvinok [30]) yields the following functions:

$$\mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}(\mathtt{l}, \mathtt{h}, 3)) = \begin{cases} 1, & \text{if} - 2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases}$$

$$\mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}(\mathtt{l}, \mathtt{h}, 0)) = \begin{cases} 2^{31} - 1 - \mathtt{l}, & \text{if } -2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{if } \mathtt{l} \geq 100 \\ 2^{32}, & \text{otherwise} \end{cases}$$

$$\mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}(\mathtt{l}, \mathtt{h}, -3)) = \begin{cases} 2^{31} + \mathtt{l}, & \text{if } -2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases}$$

$$\mathtt{C}_{\{\mathtt{l}'\}}[\mathtt{l}](\exists \mathtt{h}.\mathtt{g}(\mathtt{l}, \mathtt{h}, \mathtt{l}')) = \begin{cases} 3, & \text{if } -2^{31} < \mathtt{l} < 100 \\ 2, & \text{if } \mathtt{l} = -2^{31} \\ 1, & \text{otherwise} \end{cases}$$

From the final function we see that the maximum leakage measured by the min entropy-based metric is $\mathit{log}\ 3$ for all values of low input in the range

Listing 1.2: Listing 1.1 with specification annotations

```
 1 public class RelaxPC {
 2   public int l;
 3   private int h;
 4   /*! l | h ; !*/
 5   /*@ requires -2147483648 <= h && h < 2147483648; @*/
 6   public void check(){
 7     if (l < 100) {
 8       ...
 9     }
10   }
11 }
```

$(-2^{31}, 100)$. This restricts the choice of a suitable value for l only slightly. Computation of the maximal leakage for the Shannon and guessing entropy-based metrics requires more effort. Using the optimizers *Bonmin*[1] and *Couenne*[2] we get as result $l = 0$ which meets our intuition. Moreover, the maximum Shannon entropy leakage when choosing $l = 0$ is approximately 1, i.e. 1 bit of h is revealed. For this program, the Shannon and guessing entropy based-metric perform significantly better than the min entropy-based metric. In both cases their successive application generates a series of experiments that performs binary search to uncover the secret. □

5 Implementation and Experiments

5.1 Implementation

We implemented the approach described above on top of the KEG tool [11]. KEG is used to create failing test cases for insecure Java programs, i.e. for programs that do not adhere to a specified information flow policy. The information flow policy specification is done in source code annotations. KEG supports non-interference and delimited information release policies. For loops and (recursive) methods KEG supports loop invariants and method contracts. Beside primitive types, object types are also supported.

Listing 1.2 shows the annotated Java code from Listing 1.2. Line 4 contains a class level specification that forbids any information flow from the high variable h to the low variable l. The check method's precondition in line 5 specifies the initial knowledge about h.

The program is given to our tool which performs the analysis explained in the previous sections and illustrated in Fig. 1. Our implementation supports the optimisations described in Section 4 and outputs the corresponding optimisation problems as AMPL [12] specifications. This makes it possible to use all

[1] http://www.coin-or.org/Bonmin

[2] https://projects.coin-or.org/Couenne

Table 1: Case study statistics

File	#SP	High	Shannon entropy		Min entropy		Guessing entropy	
name	/RP	input	#RB/E	T(s)	#RB/E	T(s)	#RB/E	T(s)
PassChecker	2/2	2135451222	0/32	159	0/32	13.3	0/32	139.3
RelaxPC	4/3	-1208665253	32/31	31.7	1/32	6.9	32/31	29.4
MultiLows	6/3	395444738	32/20	22.6	1/32	7.5	32/22	24.3
ODependL	4/3	-13484756	1/1	0.9	1/1	0.2	1/1	0.3
ODependL	4/3	95464630	32/31	29.8	1/32	6.7	32/31	29.6
ODependLH	6/5	-941087637	n/a	n/a	32/1	0.7	n/a	n/a
ODependLH	6/5	23269332	n/a	n/a	1/1	0.7	n/a	n/a
LoopPlus	3/2	-552256949	n/a	n/a	1/1	0.2	n/a	n/a
LoopPlus	3/2	1707132530	n/a	n/a	32/1	1.3	n/a	n/a
EWallet	3/2	692935244	n/a	n/a	21/32	10.1	n/a	n/a

#(SP/RP): nr of **S**ymbolic **P**aths/**R**isky **P**aths

#(RB/E): nr of **R**evealed **B**its/necessary **E**xperiments T(s): Time for experiments (seconds)

optimizers that support the AMPL format. Currently, KEG uses a combination of two open source optimizers, *Bonmin* and *Couenne*, as well as the commercial optimizer *Local Solver* [5]. For model counting we use Barvinok [30]. The latter only supports counting for parametric polytopes, which restricts the use of the secret inference feature to programs whose path condition and symbolic output values are linear. This restriction does not affect KEG's other features, including leak detection (and generation of code demonstrating the leakage).

For the running example, KEG detects an information flow leak for the specified noninterference policy. In case the high variable has a value greater than 99, KEG stops after one experiment and returns $99 < \mathtt{h} < 2147483648$ as the accumulated knowledge, which is all that can be learned. However, if `h` is less or equal than 99, KEG extracts the exact value of `h` after only 31 experiments when using the Shannon or guessing entropy-based metric.

5.2 Experiment

We evaluated our approach on a sample of small, insecure programs (available at www.se.tu-darmstadt.de/research/projects/albia/download/secret-inferring/).

We assume that for any program the attacker knows nothing about the secret except that it is a 32 bit integer. Loop specifications and method contracts are supplied for programs containing unbounded loops and recursive method invocations. The tool has been configured to terminate its attack when it was either able to infer the values of the high variables, the maximum achievable knowledge has been reached (there is no way to avoid redundant low input), or the number of experiments exceeded the limit of 32. The evaluation was performed on an Intel Core i5-480M processor with 4GB RAM and Ubuntu 14.04 LTS. The results are shown in Table 1.

Discussion. Table 1 shows that using min entropy to guide experiment generation is in most cases the fastest option, but it lags often behind the other en-

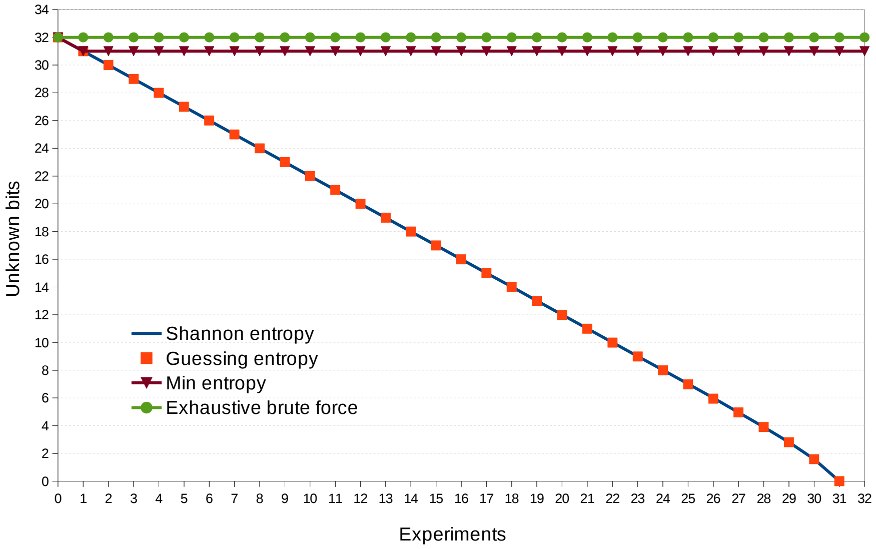

Fig. 2: Bits revealed per experiment

tropies regarding the amount of inferred information, because it considers merely the number of output values. The Shannon and guessing entropy-based metrics can only be used for analysing the programs *PassChecker*, *RelaxPC*, *MultiLows*, and *ODependL*, because only those fall into the class of programs characterized in Section 4.4. For these programs (exception *PassChecker*) the Shannon and guessing entropy-based metrics turn out to be very effective. Both reveal almost 1 bit per experiment.

Fig. 2 compares for program *RelaxPC* the number of bits revealed after each experiment for each of the supported metrics and with a simple exhaustive brute force attack (the latter could be lucky and hit the secret in one of the first 32 attempts). For this program we can see that in case of the min entropy-based metric the first experiment (which chose 0 as low level input) manages to reveal about one bit of information, namely that the secret's value is below 0 and stalls afterwards. The reason is that under the assumption of a uniform distribution the min entropy-based metric considers any possible choice of l between -2^{31} and 99 to be equally good. Consequently, the min entropy-based metric does not perform significantly better than a brute force attack. The Shannon and guessing entropy-based metrics perform best, extracting almost one bit per experiment and reveal the complete secret after 31 steps.

The program *PassChecker* is a simple password checker, leaking only whether the given input is equal to the secret or not. The amount of leakage does not depend on the low input and all entropy-based approaches perform equally bad as random experiments or exhaustive brute-force attacks.

For programs whose observable output depends on high variables (*ODependLH*, *LoopPlus* and *EWallet*), Shannon entropy and guessing entropy are practically infeasible as the range of observable values is too large. However, min entropy is still applicable and quite effective as well, as it leads to the generation of low input for execution paths on which the observable output depends on the high input. Observe that *LoopPlus* and *EWallet* contain unbounded loops and recursive method calls.

The programs *ODependL*, *ODependLH* and *LoopPlus* witness the fact that successful secret inference may also depend on the values of high variables. The reason is that in these programs the high variable influences the taken symbolic execution path and the final output values, which renders the set of reachable paths value-dependent on high variables. Hence, the quality of the generated experiments depends as well on the high variables.

6 Related Work

An information-theoretic model for an adaptive side-channel attack is proposed in [18]. The idea of the attacker strategy is to choose at each step the query that minimizes the remaining entropy. Even though this greedy heuristic attack strategy is similar to our guided experiments approach, it requires to enumerate all possible queries to choose the best one, which is rather expensive. Our approach differs in the sense that we quantify the potential leakage as a function of low input, and hence, we can make use of many available, efficient optimization tools to find the optimal input value.

Pasareanu et al. [23] propose a non-adaptive side-channel attack to find low input that maximizes the amount of leaked information. To find optimal low input, they solve a number of Max-SMT problems whose formalisation is based on path conditions and user-defined cost models. In contrast to our approach, only path conditions are considered, but not symbolic states. Hence, they cannot measure leakage caused by explicit information flow. The authors of [15] define a *quantitative policy* which specifies an upper bound for permitted information leakage. The model checker CBMC is used to generate low input that triggers a violation of the policy. Both of [15,23] use channel capacity as their leakage metric which is the worst case over all *prior distributions* over high inputs. Low input is generated with the aim to maximize the number of equivalence classes on high inputs. The size of the individual class is not taken into account. Hence, they are less precise than our approach that takes into account prior distributions. Their generated low input often is not the optimal one: for example, in case of Listing 1.2, we are able to generate a sequence of low inputs for `l`, each of which extracts nearly 1 bit of information, allowing to find the exact secret after 31 experiments. Their approach can only return a single, *arbitrary* input for $\mathtt{l} \in (-2^{31}, 100)$, hence, using it for an attack would not perform better than brute force (see discussion in Section 5.2). Both approaches require a bound on the number of loop iterations or the recursion depth, whereas we can make use of specifications to deal with unbounded loops and recursion.

Low input as a parameter of quantitative information flow (QIF) analysis is also addressed in [22,31]. In [31], the authors only analyze the bounding problem of QIF for low input, but do not provide a method to determine a bound for the leakage. The authors of [22] model the program with low input as a set of information channels, where each channel corresponds to a specific value of low input. While considering that the leakage depends on low input, they do not discuss how to find the input maximizing the leakage.

Symbolic execution as a static analysis technique is used in several information flow analyses, both qualitative [10,4,24] and quantitative [17,25]. In [17] a precise quantitative information flow analysis based on calculating cardinalities of equivalent classes is presented. The author assumes an optimally chosen set of experiments, but does not describe how to construct such a set.

The authors of [7,8] model the attacker's knowledge about the secret as belief and show how to update the inferred secret after each experiment. In [3], the authors briefly discuss the correlation between the set of experiments and an attacker's knowledge about the secret. However, none of these papers describes how to construct an optimal experiment set that maximizes the leakage. Other approaches in quantitative information flow [25,21,28,19] do not address low input in their analyses and consider only channel capacity with the same drawbacks as discussed earlier.

7 Conclusion and Future Work

We presented an approach to automatically infer secrets leaked by an information flow-insecure program. It features a novel, adaptive algorithm that (i) combines static and dynamic analysis, (ii) uses leakage metrics that *depend on low input* (which, to the best of our knowledge, sets it apart from any existing work) to guide experiment generation and (iii) provides a logic characterisation of the search space for the secret that can be put into a model finder to extract the secrets. A first implementation of the approach that can deal with programs containing loops and recursive methods has been presented. The viability of the method has been demonstrated with a number of small, but representative benchmark programs that clearly illustrate its potential and its current limitations.

In the future we plan to integrate specification generation techniques [13] to reduce the need for user-provided specification in presence of loops and recursion. We will also look into a possible extension of our approach to enable the secret inference engine to deal with programs that have non-linear path conditions or symbolic output values.

Currently we make the assumption that the prior distribution over high inputs is uniform. In ongoing work we aim to support user-defined distributions that could, for instance, be taken from data about password leaks.[3]

We also intend to perform a larger case study and plan to apply our tool to real-world programs, for which we will investigate further speed improvements.

[3] https://haveibeenpwned.com

References

1. M. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring Information Leakage Using Generalized Gain Functions. In *Computer Security Foundations Symp. (CSF), 2012 IEEE 25th*, pages 265–279, June 2012.
2. M. S. Alvim, A. Scedrov, and F. B. Schneider. When Not All Bits Are Equal: Worth-Based Information Flow. In M. Abadi and S. Kremer, editors, *Principles of Security and Trust*, volume 8414 of *LNCS*, pages 120–139. Springer, 2014.
3. M. Backes, B. Kopf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *30th Symp. on Security and Privacy*, pages 141–153, 2009.
4. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proc. of 17th IEEE Computer Security Foundations Workshop*, pages 100–114, June 2004.
5. T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua. Localsolver 1.x: a black-box local-search solver for 0-1 programming. *4OR*, 9:299–316, 2011.
6. D. Clark, S. Hunt, and P. Malacaria. A Static Analysis for Quantifying Information Flow in a Simple Imperative Language. *J. Comput. Secur.*, 15(3):321–371, 2007.
7. M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18), Aix-en-Provence, France*, pages 31–45. IEEE Computer Society, 2005.
8. M. R. Clarkson, A. C. Myers, and F. B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, 2009.
9. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security.* IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.
10. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proc. of the Second Intl. Conf. on Security in Pervasive Computing*, SPC'05, pages 193–209. Springer, 2005.
11. Q. H. Do, R. Bubel, and R. Hähnle. Exploit Generation for Information Flow Leaks in Object-Oriented Programs. In H. Federrath and D. Gollmann, editors, *ICT Systems Security and Privacy Protection*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 401–415. Springer, 2015.
12. D. M. Gay. The AMPL Modeling Language: An Aid to Formulating and Solving Optimization Problems. In *Numerical Analysis and Optimization*, pages 95–116. Springer, 2015.
13. R. Hähnle, N. Wasser, and R. Bubel. Array abstraction with symbolic pivots. In E. Ábrahám, M. Bonsangue, and E. B. Johnsen, editors, *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *LNCS*, pages 104–121. Springer, 2016.
14. M. Hentschel, R. Hähnle, and R. Bubel. Visualizing unbounded symbolic execution. In M. Seidl and N. Tillmann, editors, *Tests and Proofs, 8th International Conference, York, UK*, volume 8570 of *LNCS*, pages 82–98. Springer, 2014.
15. J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 261–269, New York, NY, USA, 2010. ACM.
16. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.
17. V. Klebanov. Precise quantitative information flow analysis—a symbolic approach. *Theoretical Computer Science*, 538:124–139, 2014.

18. B. Köpf and D. Basin. An Information-theoretic Model for Adaptive Side-channel Attacks. In *Proc. of the 14th ACM Conf. on Computer and Communications Security*, CCS '07, pages 286–296. ACM, 2007.
19. P. Malacaria and H. Chen. Lagrange Multipliers and Maximum Information Leakage in Different Observational Models. In *Proc. of the 3rd ACM SIGPLAN Workshop on Prog. Languages and Analysis for Security*, PLAS '08, pages 135–146. ACM, 2008.
20. J. L. Massey. Guessing and entropy. In *Proc. on IEEE Intl. Symp. on Information Theory*, Jun 1994.
21. Z. Meng and G. Smith. Calculating Bounds on Information Leakage Using Two-bit Patterns. In *Proc. of the ACM SIGPLAN 6th Workshop on Prof. Languages and Analysis for Security*, PLAS '11, pages 1:1–1:12. ACM, 2011.
22. T. M. Ngo and M. Huisman. Quantitative Security Analysis for Programs with Low Input and Noisy Output. In *Proc. of the 6th Intl. Symp. on Engineering Secure Software and Systems*, volume 8364 of *ESSoS 2014*, pages 77–94. Springer, 2014.
23. C. S. Pasareanu, Q. Phan, and P. Malacaria. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal*, pages 387–400. IEEE Computer Society, 2016.
24. Q.-S. Phan. Self-composition by Symbolic Execution. In A. V. Jones and N. Ng, editors, *Imperial College Computing Student Workshop*, volume 35 of *OASIcs*, pages 95–102. Schloss Dagstuhl, 2013.
25. Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.
26. D. E. Robling Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
27. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, Oct. 2009.
28. G. Smith. On the Foundations of Quantitative Information Flow. In *Proc. of the 12th Intl. Conf. on Foundations of Software Science and Computational Structures*, FOSSACS '09, pages 288–302. Springer, 2009.
29. G. Smith. Quantifying information flow using min-entropy. In *8th Intl. Conf. on Quantitative Evaluation of Systems*, QEST 2011, pages 159–167. IEEE Computer Society, 2011.
30. S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.
31. H. Yasuoka and T. Terauchi. On Bounding Problems of Quantitative Information Flow. *J. Comput. Secur.*, 19(6):1029–1082, 2011.

The appendix is just for the convenience of the reviewers and will be removed for the final version including all references to it from the paper.

A Proof Outlines

A.1 Lemma 1

Proof. Consider two arbitrary concrete values $\overline{h}_1, \overline{h}_2 \in \mathbb{H}$ for which $K^E(\overline{h}_1)$ and $K^E(\overline{h}_2)$ hold. Let $\overline{o}_1, \overline{o}_2 \in \mathbb{O}_D(\overline{l})$ be output values observed in the final state of two program runs taking $(\overline{l}, \overline{h}_1)$ and $(\overline{l}, \overline{h}_2)$ as input values, respectively. We will prove that $\overline{o}_1 = \overline{o}_2$. Assume that $\overline{o}_1 \neq \overline{o}_2$. Because $\overline{l}$ is i-matched and $K^E(\overline{h}_1)$, $K^E(\overline{h}_2)$ hold, we get as direct consequence of Definition 7 that $pc_i[\overline{l}, \overline{h}_1/L, H]$ and $pc_i[\overline{l}, \overline{h}_2/L, H]$ are true. This means that the two concrete runs with $(\overline{l}, \overline{h}_1)$ and $(\overline{l}, \overline{h}_2)$ as input values correspond to the symbolic execution path i, and hence, by the correctness of symbolic execution that $\overline{o}_1 = f_i^O[\overline{l}, \overline{h}_1/L, H]$ and $\overline{o}_2 = f_i^O[\overline{l}, \overline{h}_1/L, H]$. By assumption $\overline{o}_1 \neq \overline{o}_2$, i.e., $f_i^O[\overline{l}, \overline{h}_1/L, H] \neq f_i^O[\overline{l}, \overline{h}_1/L, H]$. Consequently, we have

$$pc_i[\overline{l}, \overline{h}_1/L, H] \wedge pc_i[\overline{l}, \overline{h}_2/L, H] \wedge f_i^O[\overline{l}, \overline{h}_1/L, H] \neq f_i^O[\overline{l}, \overline{h}_1/L, H] \equiv \mathit{true}$$

Thus formula $\mathit{Leak}(i, i)$ (see (1)) is satisfied by $\overline{l}, \overline{h}_1, \overline{h}_2$, which means that i is a risky path, which contradicts the assumption of the lemma that i is not a risky path. Hence $\overline{o}_1 = \overline{o}_2$ and this lemma is proven. □

A.2 Theorem 1

By Lemma 1 we know that for a given $\overline{l} \in \mathbb{L}$ is i-matched with $i \notin \mathit{Risk}$, all program runs produce the same output $\overline{o} \in \mathbb{O}$ for any high input $\overline{h} \in \mathbb{H}$ for which $K^E(\overline{h})$ holds.

Hence we have only to prove that

$$\forall \overline{h}.(K^E(\overline{h}) \rightarrow \mathit{Info}_{\langle \overline{l}, \overline{o} \rangle}(\overline{h})) \tag{10}$$

Let $\overline{h}_0$ be an arbitrary but fixed value in $\mathbb{H}$. We have to show that

$$K^E(\overline{h}_0) \rightarrow \mathit{Info}_{\langle \overline{l}, \overline{o} \rangle}(\overline{h}_0) \tag{11}$$

is true.

Case 1: If $K^E(\overline{h}_0)$ is false then (11) is trivially true (semantics of implication) and we are done.

Case 2: We can now assume that $K^E(\overline{h}_0)$ is true. Because $\overline{l}$ is i-matched (assumption of the theorem) we have $\mathit{Info}_{\langle \overline{l}, \overline{o} \rangle}(\overline{h}_0) \equiv \mathit{InfoPath}_i(\overline{l}, \overline{h}_0, \overline{o}) \equiv$

$$pc_i[\overline{l}, \overline{h}_0/L, H] \wedge \bigwedge_{v \in O} o_v = f_i^v[\overline{l}, \overline{h}_0/L, H]$$

where $\overline{o} = \{o_v | v \in O\}$.

- The validity of $pc_i[\bar{l}, \overline{h}_0/L, H]$ follows directly from Definition 7 and our case assumption ($K^E(\overline{h}_0)$ holds).
- The second conjunct $\bigwedge_{v \in O} o_v = f_i^v[\bar{l}, \overline{h}_0/L, H]$ is a direct consequence of the correctness of symbolic execution: The $\overline{o}$ are the result of running program p with input $\bar{l}, \overline{h}_0$, hence, given the correctness of symbolic execution the symbolic output values must evaluate to the same concrete values.

□

A.3 Theorem 2

We can rewrite the definition of $\mathit{Info}_{\langle \bar{l}, \bar{o} \rangle}(H)$ ($\overline{o} = \{o_v | o_v \in \mathbb{O}_{\overline{l}}, v \in O\}$) (from (6) and (7)) simply to

$$\big(\bigvee_{i \in R^E} \mathit{InfoPath}_i(\bar{l}, H, \bar{o})\big) \vee \big(\bigvee_{i \notin R^E \wedge 1 \leq i \leq N_p} \mathit{InfoPath}_i(\bar{l}, H, \bar{o})\big)$$

To prove Theorem 2, we prove that for all $i \notin R^E$ and $1 \leq i \leq N_p$

$$K^E(H) \wedge \mathit{InfoPath}_i(\bar{l}, H, \bar{o}) \tag{12}$$

is unsatisfiable (i.e., equivalent to *false*).
Let $i0 \notin R^E$ be an arbitrary but fixed unreachable path. $\mathit{InfoPath}_{i0}(\bar{l}, H, \bar{o})$ is defined as

$$pc_{i0}[\bar{l}/L] \wedge \bigwedge_{v \in O} o_v = f_{i0}^v[\bar{l}/L]$$

Because $i \notin R^E$, by Definition 8, we have $K^E(H) \wedge pc_{i0}$ is unsatisfiable, hence

$$K^E(H) \wedge \mathit{InfoPath}_{i0}(\bar{l}, H, \bar{o})$$

is unsatisfiable. which proves (12) and therewith Theorem 2.

□

A.4 Theorem 3

Shannon entropy According to (3), we have

$$\mathtt{ShEL_p}(L) = \mathcal{H}(O_{out}(L)) \underset{\text{Def. 1}}{=} - \sum_{\overline{o} \in \mathbb{O}_D(L)} P(O_{out}(L) = \overline{o}) \log P(O_{out}(L) = \overline{o})$$

By definition, $O_{out}(L)$ is the value of O observed after running program with low input L and H_{in}. We have by the law of total probability

$$P(O_{out}(L) = \overline{o}) = \sum_{\overline{h} \in \mathbb{H}} P(H_{in} = \overline{h}) P(O_{out}(L) = \overline{o} | H_{in} = \overline{h})$$

Under the assumption that $K^E(H)$ is correct, from Corollary 1 and Theorem 2, we only consider values of L that satisfy $\mathit{InRisk}(L)$ (to avoid redundant experiments) and take into account only reachable paths for $K^E(H)$. For any $\bar{l} \in \mathbb{L}$, we have:

$$P(O_{out}(\bar{l}) = \overline{o} | H_{in} = \overline{h}) = \\ \begin{cases} 1, & \text{if } \overline{h} \in \mathit{Sat}(K^E(H) \wedge \mathit{InRisk}(\bar{l}) \wedge \bigvee_{i \in R^E} pc_i[\bar{l}/L] \wedge \bigwedge_{v \in O} o_v = f_i^v[\bar{l}/L]) \\ 0, & \text{otherwise} \end{cases}$$

where $\overline{o} = \{o_v | v \in O\} \in \mathbb{O}_D(\bar{l})$.
Because H_{in} has uniform distribution, we have:

$$P(H_{in} = \overline{h}) = \begin{cases} \frac{1}{S_E}, & \text{if } \overline{h} \in \mathit{Sat}(K^E(H)) \\ 0, & \text{otherwise} \end{cases}$$

where $S_E = |\mathit{Sat}(K^E(H))|$ (defined in Section 4.2). Recall that $g(L, H, \overline{o}) = K^E(H) \wedge \mathit{InRisk}(L) \wedge \bigvee_{i \in R^E} pc_i \wedge \bigwedge_{v \in O} o_v = f_i^v$, we have

$$\begin{aligned} \forall \bar{l} \in \mathbb{L}. P(O_{out}(\bar{l}) = \overline{o}) &= \frac{1}{S_E} \sum_{\overline{h} \in \mathit{Sat}(K^E(H))} P(O_{out}(\bar{l}) = \overline{o} | H_{in} = \overline{h}) \\ &= \frac{|\mathit{Sat}(\mathtt{g}(\bar{l}, H, \overline{o}))|}{S_E} \end{aligned} \tag{13}$$

Definition 9, (13) give us

$$P(O_{out}(L) = \overline{o}) = \frac{\mathtt{C}_H[L](\mathtt{g}(L, H, \overline{o}))}{S_E} \tag{14}$$

Because $\sum_{\overline{o} \in \mathbb{O}_D(L)} P(O_{out}(L) = \overline{o}) = 1$, we have

$$\sum_{\overline{o} \in \mathbb{O}_D(L)} \mathtt{C}_H[L](g(L, H, \overline{o})) = S_E$$

Shannon entropy-based leakage is then computed as:

$$\begin{aligned} \mathtt{ShEL_p}(L) &= - \sum_{\overline{o} \in \mathbb{O}_D(L)} \left(\frac{\mathtt{C}_H[L](g(L, H, \overline{o}))}{S_E} \mathit{log}\left(\frac{\mathtt{C}_H[L](g(L, H, \overline{o}))}{S_E} \right) \right) \\ &= \mathit{log}(S_E) - \frac{1}{S_E} \sum_{\overline{o} \in \mathbb{O}_D(L)} \big(\mathtt{C}_H[L](g(L, H, \overline{o})) \mathit{log}(\mathtt{C}_H[L](g(L, H, \overline{o}))) \big) \end{aligned}$$

Min entropy According to (4), the min entropy-based leakage of deterministic program p with uniform distribution of H_{in} can be computed as below:

$$\mathtt{MEL_p}(L) = \log |\mathbb{O}_D(L)|$$

Under the assumption that the attacker's knowledge of H is correct, we have for any low input $\overline{l} \in \mathbb{L}$, $\overline{o} \in \mathbb{O}_D(\overline{l})$ if and only if there exists a concrete value $\overline{h}_0 \in Sat_H(K^E(H))$ such that program p taking $\overline{h}_0$ and $\overline{l}$ as high and low input respectively produces observable output $\overline{o}$. Thus we have

$$\begin{aligned}\forall \overline{l} \in \mathbb{L}.\ \mathtt{MEL_p}(\overline{l}) = log|\mathbb{O}_D(\overline{l})| = \\ log|Sat(\exists \overline{h}.K^E(\overline{h}) \wedge \bigvee_{i \in R^E} pc_i[\overline{l}, \overline{h}/L, H] \wedge \bigwedge_{v' \in O'} v' = f_i^v[\overline{l}, \overline{h}/L, H])| \\ = log|Sat(\exists \overline{h}.\mathtt{g}(\overline{l}, \overline{h}, O'))|\end{aligned}$$

Hence by definition 9, $\mathtt{MEL_p}(L) = \log(\mathtt{C}_{O'}[L](\exists \overline{h}.g(L, \overline{h}, O')))$

Guessing entropy According to (5), we have:

$$\mathtt{GEL_p}(L) = \mathcal{G}(H_{in}) - \mathcal{G}(H_{in}|O_{out}(L))$$

Under the assumption that H_{in} has uniform distribution, we have

$$\forall i \in [1, S_E].\ P(H_{in} = \overline{h}_i) = \frac{1}{S_E}$$

thus we have

$$\mathcal{G}(H_{in}) \underset{\text{Def. 3}}{=} \sum_{1 \leq i \leq S_E} i \cdot P(H_{in} = \overline{h_i}) = \frac{1}{S_E} \sum_{1 \leq i \leq S_E} i = \frac{S_E(S_E+1)}{2S_E} = \frac{S_E+1}{2}$$

by Definition 3 we have

$$\mathcal{G}(H_{in}|O_{out}(L)) = \sum_{\overline{o} \in \mathbb{O}_D(L)} P(O_{out}(L) = \overline{o}) \mathcal{G}(H_{in}|O_{out}(L) = \overline{o}) \tag{15}$$

where $\mathcal{G}(H_{in}|O_{out}(L) = \overline{o}) = \sum_{1 \leq i \leq S_E} i \cdot P(H_{in} = \overline{h_i}|O_{out}(L) = \overline{o})$. For any $\overline{l} \in \mathbb{L}$, we have

$$P(H_{in} = \overline{h_i}|O_{out}(\overline{l}) = \overline{o}) \underset{\text{Bayes}}{=} \frac{P(O_{out}(\overline{l}) = \overline{o}|H_{in} = \overline{h_i})P(H_{in} = \overline{h_i})}{P(O_{out}(\overline{l}) = \overline{o})}$$

Similarly to proof for the case of Shannon entropy, we have

$$\forall \overline{l} \in \mathbb{L}.\ P(H_{in} = \overline{h_i}|O_{out}(\overline{l}) = \overline{o}) = \begin{cases} \frac{1}{|Sat(\mathtt{g}(\overline{l}, H, \overline{o}))|}, & \text{if } \overline{h_i} \in Sat(\mathtt{g}(\overline{l}, H, \overline{o})) \\ 0, & \text{otherwise} \end{cases}$$

Hence we have (by Definition 9)

$$\begin{aligned}\mathcal{G}(H_{in}|O_{out}(L) = \overline{o}) = \frac{1}{\mathtt{C}_H[L](g(L, H, \overline{o}))} \sum_{1 \leq i \leq \mathtt{C}_H[L](g(L,H,\overline{o}))} i \\ = \frac{\mathtt{C}_H[L](g(L, H, \overline{o})) + 1}{2}\end{aligned} \tag{16}$$

Using formula (14) together with (15) and (16) we get

$$\mathcal{G}(H_{in}|O_{out}(L)) = \sum_{\overline{o} \in \mathbb{O}_D(L)} \frac{\mathtt{C}_H[L](g(L,H,\overline{o}))(\mathtt{C}_H[L](g(L,H,\overline{o}))+1)}{2S_E}$$

Hence, guessing entropy-based leakage can be computed as follows:

$$\mathtt{GEL}_{\mathtt{p}}(L) = \frac{S_E+1}{2} - \frac{1}{2S_E} \sum_{\overline{o} \in \mathbb{O}_D(L)} \big(\mathtt{C}_H[L](g(L,H,\overline{o}))(\mathtt{C}_H[L](g(L,H,\overline{o}))+1)\big)$$

□