Efficient SAT-Based Pre-image Enumeration for Quantitative Information Flow in Programs

Alexander Weigl(✉)

Karlsruhe Institute of Technology,
Am Fasanengarten 5, 76131 Karlsruhe, Germany
weigl@kit.edu

Abstract. Quantitative Information Flow Analysis (QIF) measures the loss of an attacker's uncertainty about the confidential information (pre-image) inside a software system after observing the system outputs (image). In this paper, we supplement the SAT-based QIF analysis for deterministic and terminating C programs, by introducing three algorithms for counting the pre-images and images, which utilizes advantages of incremental SAT solvers. Our tool SHARPPI is competitive to MQL, QUAIL and CHIMP. An implementation is provided under http://formal.iti.kit.edu/sharpPI.

1 Introduction

Under Quantitative Information Flow Analysis (QIF) we subsume techniques and approaches to measure information flow in software systems. The information flow is an influence between two program variables and is usually described with entropy, which is a measure for the uncertainty about an information. The typical application for QIF is associated with an attacker, who tries to reduce its uncertainty over secrets, e.g. passwords or pin numbers, of a system by viewing the observable information. The desired property of a system is the absence of information flow between the secret and observable information, hence the attacker is not able to learn anything about the secret information. This *non-interference* property is not always achievable in practice. For example, the usual login on web pages leaks a bit information over the users and passwords with every login attempt. QIF's motivation is to provide a metric for the assessment and comparison of information flows between different implementations. A smallest possible information flow to the observable information is desired (information leakage), because it leaves behind the highest uncertainty about the secret information for the attacker.

This work bases on [5], which introduces an approach for calculating the min entropy of information flow in C programs. The authors use CBMC [6] to

A. Weigl—This work was supported by the DFG (German Research Foundation) in Priority Programme Reliably Secure Software Systems (RS3) – DFG Priority Programme 1496. Thanks to Vladimir Klebanov for feedback during the creation of this paper and Laurent Simon for a dessert in Lisbon.

G. Livraga et al. (Eds.): DPM and QASA 2016, LNCS 9963, pp. 51–58, 2016.
DOI: 10.1007/978-3-319-47072-6_4

generate a formula in conjunctive normal form of a program and apply model counting on the propositional formula to enumerate all possible observable information. This SAT-based approach has some advantages. Every performance gain in #SAT or SAT solver is directly applicable. We support real (bounded) C programs, but the input language is changeable, as long there is a translation into a CNF formula.

Contributions. We supplement the SAT-based approach from [5] with three different algorithms UnGuided, Bucket-wise and Sync (Sect. 2) for counting the secret state (pre-image) and corresponding observable output (image) for the calculation of the Shannon entropy. We compare the our algorithms to other QIF analysis tools with a part of the case study in [1] (Sect. 3). An implementation is provided.

Foundations. We give a brief overview to foundations of QIF analysis. A detailed overview is in [5]. We investigate the degree of influence during the program execution between the secret information (*high*) at the start state and the observable information (*low*) at the final state. For measuring, we model this this influence as a function π, that maps from high value $\mathcal{H}$ to the low output value $\mathcal{O}$:

$$\pi\colon \mathcal{H} \to \mathcal{O}.$$

With this model, we omit the local variables, which have a fix value at the start state given by the program semantics, whereas the high variables have an arbitrary value (for the attacker unknown). For clarification, $\mathcal{H}$ is the domain, $\mathcal{O}$ the codomain of the function π, the images $o \in \mathcal{O}$ are the output values and the pre-images $\pi^{-1}(o) \subseteq \mathcal{H}$, defined as $\pi^{-1}(o) = \{h \in \mathcal{H} \mid \pi(h) = h\}$. For deterministic programs the pre-images are disjoint. For convenience, we silently lift multiple high or low variables to tuples.

We use CBMC for the translation of a program into a corresponding propositional formula φ over a signature Σ in conjunctive normal form (CNF). The formula φ represents a program, s.t. every model of φ is a valid program trace. Each variable is encoded by a set propositional variables. We are interested into the signature $\mathbb{H} \subseteq \Sigma$ that encodes the high variable, and $\mathbb{O} \subseteq \Sigma$ the low variable. By projection on these both signatures, we obtain the function π. $\varphi\big|_{\Delta}$ denotes the projection of φ to the signature $\Delta \subseteq \Sigma$. The projection $\varphi\big|_{\Delta}$ is the strongest Δ-formula, that is entailed by φ if interpreted over Σ. The projection of a model m is obtained by dropping every variable $v \notin \Delta$. A model m of φ contains the encoded values for high and low variables, that we retrieve by projection $m\big|_{\mathbb{H}}$, resp. $m\big|_{\mathbb{O}}$.

Under the assumption of termination, determinism and with uniform distribution of the input values, we the conditional Shannon entropy [5,7].

Definition 1 (Cond. Shannon Entropy for Deterministic Programs).

$$H(\mathtt{X}|\mathtt{Y}) = \frac{1}{\#(X)} \sum_{y \in \mathcal{Y}} \#(\pi^{-1}(y)) \log \#(\pi^{-1}(y))$$

The conditional Shannon entropy only depends on the sizes of the pre-images and images and is invariant on the their order. In the remaining sections of this paper, we always reference to this conditional version of the Shannon entropy.

2 Counting Algorithms for (Pre-)Images

We introduce the three algorithms UnGuided, Bucket-wise and Sync with different ways of counting, which are special instance of the model counting problem with projection #SAT-p. We need to count with projection to the signature of either the high input variable $\mathbb{H} \subseteq \Sigma$ or the low output variable $\mathbb{O} \subseteq \Sigma$. We want to utilize the working principals of the incremental SAT solver to achieve an efficient counting of the images and the pre-images.

The algorithms produce a histogram $\mathit{Hist}\colon \mathcal{O} \to \mathbb{N}$ (Fig. 1), which associates every possible output of π to the size of its pre-image: $\mathit{Hist}(o) = \#(\pi^{-1}(o))$. We denote o's place in an histogram as its *bucket*.

Input and Output of the Algorithms. The algorithms have three input parameters: a propositional formula φ over signature Σ in conjunctive normal form (CNF), the signature of the high input variable $\mathbb{H} \subseteq \Sigma$ and the signature $\mathbb{O} \subseteq \Sigma$ of the low output variable.

The result of the algorithms is the precise histogram *Hist*. Furthermore, the algorithms Bucket-wise and Sync are able to decide whether all inputs values of a pre-image are counted, represented by the function $\mathit{closed}\colon \mathcal{O} \to \{\mathit{true}, \mathit{false}\}$. If $\mathit{closed}(o)$ is true, then the bucket $\mathit{Hist}(o)$ is final. Histogram *Hist* is initialized with zeros, resp. *closed* with *false* entries.

Used Functions. The algorithms are based upon the decision problem (SAT) for satisfiability of propositional formula φ. SAT(φ) denotes a call to the SAT solver with a CNF formula. The returned value is either a model m or $\perp$ to signal unsatisfiability. We can supply an assumption a, denoted as SAT($a \Rightarrow \varphi$). An assumption is a partial assignment of variables, which constrains the SAT solver to find a model that ensures the assumption's assignments.

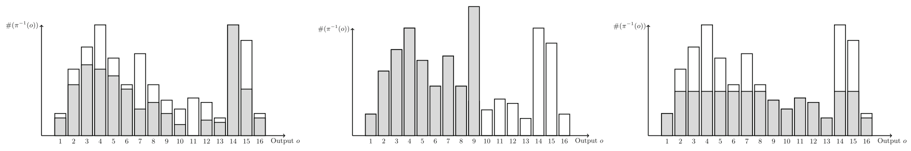
Our counting algorithms work by adding blocking clauses to exclude already found values of input or output variables. For the construction of blocking clauses, we define the function $\mathit{block}(\varphi, m, \Delta)$, which takes a CNF formula φ, a model m and a signature Δ. The function returns a new CNF formula φ', s. t. the projected model $m\big|_{\Delta}$ is not a part of any model of φ'.

Implementation. An efficient implementation of the algorithms UnGuided, Bucket-wise and Sync requires an incremental SAT solver, which offers two operations: (a) appending of new clauses to CNF formula and (b) finding a satisfying assignment under an assumption. An incremental SAT solver reuses information from previous runs. Hence, subsequent calls to solver take less time. In the concrete implementation, we reuse the SAT solver instance and block a model by adding the blocking clause to the instance. This detail is omitted in shown version of the algorithms to attain a better readability.

Brief Overview of the Algorithm. We give here a brief overview of the algorithms, cf. Fig. 1. The Algorithm UnGuided iterates over all models in the order determined by the SAT solver. The occurrence of corresponding pairs of input and output values may be chaotic or random (Fig. 1a). The Algorithm Bucket-wise counts a pre-image for a particular image, before it starts with a further pre-image (Fig. 1b). In each iteration, the Algorithm Sync searches for one new input value for every image, until all input values are found (Fig. 1c). The experiments and discussion takes place in Sect. 3.

Unguided Counting. The Algorithm UnGuided is the logical extension of the algorithm given in [5, Fig. 2]. The choice of the next model is left to the (incremental) SAT solver, which we give the most degrees of freedom to reuse the most information from the previous runs.

In comparison to [5], both implementations iterate over the sets of models $models(\varphi|_{\Delta})$, but our implementation does not collect the models. Instead, we extract the output value $m|_{\mathbb{O}}$, and increase the corresponding bucket in the histogram $Hist(m|_{\mathbb{O}})$ for each found model. Due to the determinism of program, there is no other output value for the last found input value $m|_{\mathbb{H}}$. Hence, we block the input value from further occurence to prevent a double counting. The function call $block(\varphi, m, \mathbb{H})$ returns a clause set that prohibits the assignment of input values in the further calls of the SAT solver. The algorithm does not provide information if a bucket is closed (Fig. 2).



(a) UnGuided (b) Bucket-wise (c) Sync

Fig. 1. Graphical representation of the effects of different algorithms on distribution of the size of the input partitions during counting. The gray bar represents the counted elements of the bucket, whereas the white bar symbolizes the true, but unknown, part.

```
Input: A propositional formula $\varphi$ over $\Sigma$, a signature $\mathbb{O} \subseteq \Sigma$ representing the output
       variable, and $\mathbb{H} \subseteq \Sigma$ for the input variable
Output: Histogram $\forall o \in \mathcal{O}$: $Hist(o) = \#(\pi^{-1}(o))$
begin
    while $m := \mathrm{SAT}(\varphi)$ do
        $Hist(m|_{\mathbb{O}}) := Hist(m|_{\mathbb{O}}) + 1$
        $\varphi := block(\varphi, m, \mathbb{H})$
    end
end
```

Fig. 2. Algorithm UnGuided iterates unstructured over every model.

```
Input: A propositional formula $\varphi$ over $\Sigma$, a signature $\mathbb{O} \subseteq \Sigma$ representing the output
       variable, and $\mathbb{H} \subseteq \Sigma$ for the input variable
Output: Histogram $\forall o \in \mathcal{O}\colon \mathit{Hist}(o) = \#(\pi^{-1}(o))$
begin
    while $m := \mathrm{SAT}(\varphi)$ do
        $o \leftarrow m|_{\mathbb{O}}$
        do
            $\mathit{Hist}(m|_{\mathbb{H}}) := \mathit{Hist}(m|_{\mathbb{H}}) + 1$
            $\varphi := \mathit{block}(\varphi, m, \mathbb{H})$
        while $m := \mathrm{SAT}(o \Rightarrow \varphi)$
        $\mathit{closed}(o) := \mathit{true}$
        $\varphi := \mathit{block}(\varphi, m, \mathbb{O})$
    end
end
```

Fig. 3. Bucket-wise counting (Bucket-wise) tries to fill a bucket, before it descends a new bucket.

```
Input: A propositional formula $\varphi$ over $\Sigma$, a signature $\mathbb{O} \subseteq \Sigma$ representing the output
       variable, and $\mathbb{H} \subseteq \Sigma$ for the input variable
Output: Histogram $\forall o \in \mathcal{O}\colon \mathit{Hist}(o) = \#(\pi^{-1}(o))$ and $\mathit{closed}\colon \mathcal{O} \to \mathbb{B}$
begin
    $O := \{m|_{\mathbb{O}} \mid m \in \mathrm{models}(\varphi)\}$
    $\mathit{finished} := \mathit{false}$
    while $\neg\mathit{finished}$ do
        $\mathit{finished} := \mathit{true}$
        for $o \in O \wedge \neg\mathit{closed}(o)$ do
            if $m := \mathrm{SAT}(o \Rightarrow \varphi)$ then
                $\mathit{Hist}(m|_{\mathbb{O}}) := \mathit{Hist}(m|_{\mathbb{O}}) + 1$
                $\varphi := \mathit{block}(\varphi, m, \mathbb{H})$
                $\mathit{finished} := \mathit{false}$
            else
                $\mathit{closed}(o) := \mathit{true}$;
            end
        end
    end
end
```

Fig. 4. Algorithm Sync, synchronized counting of every bucket, by finding (1) all reachable output values and (2) iterating over all output values and increasing its bucket, until all pairs of input and output values are reached.

Bucket-wise Counting. The idea behind the Algorithm Bucket-wise (Fig. 3) is to fix an output value $o \in \mathcal{O}$ and exhaustively count all input values in the corresponding pre-image. We guide the SAT solver through the iteration over the models by setting assumptions. We hope the focus on one pre-image increases the performance of the SAT solver, because the SAT solver *only* needs to find another input value, after it has discovered a similar input and output value relation.

The Algorithm Bucket-wise starts with SAT(φ) to find the first relation between an input value and output value of function π. In the next step, we fix the output value $o = m|_{\mathbb{O}}$ and use o as the assumption in further SAT applications SAT($o \Rightarrow \varphi$) until the φ is unsatisfiable under this assumption, so the pre-image is counted exhaustively and the bucket is closed. The Line 9 in Fig. 3 blocks an exhaustively explored output value o. Blocking the output value o is not required, because all possible input values of o have been blocked. We block o to give more explicit information to the SAT solver. We repeat this procedure, until all output values are blocked and φ becomes unsatisfiable.

Synchronized Counting. An uniform distribution of input values over the images is the best case for an attacker. This idea motivates the Algorithm SYNC to maintain an uniform distribution as long as possible, as the lower bound of the Shannon entropy.

The Algorithm SYNC (Fig. 4) starts with calculation of the reachable output values in the π's codomain. The main part is a fix point algorithm, which stops if φ becomes unsat during the counting iff all pre-images are counted. The inner for-loop iterates over all output values O, that might have an undiscovered corresponding input value. If a model m is found, then we increase the corresponding bucket and block the input value; the fix point isn't reached. If no model is found, the bucket is closed.

The concrete implementation integrates the search for the reachable output values (Line 2) and assigns each blocking clause of an output value a fresh label literal for selecting the desired output value.

3 Experiment and Discussion

This experiment serves for the comparison of our tool SHARPPI with other state-of-the-art tool for QIF analysis. We use the "all houses" scenario inside the "Smart Grid" case study [1]. The Fig. 5 gives the program in C. This scenario describes an attacker, who wants to gain knowledge about occupied houses of a city block, which contains N houses, evenly split up in three different sizes. Every house size has a specific consumption. The attacker is able to observe the *global consumption* of the block, which is sum of every consumption of every occupied house. In the following we consider the case B, with the 1 unit for small, 3 units for medium and 5 units for large consumption.

```
int allhouses(bool presence[N]){
    int low = 0;
    for(int i = 0; i < N; i++) {
        if (presence[i]) {
            if      (i< N/3)      { low = low + SMALL; }
            else if (i< 2*(N/3)) { low = low + MEDIUM;}
            else                  { low = low + LARGE; }
        }}
    return low;}
```

Fig. 5. "All houses" case study from [1] given as C program. N is the number of all houses.

Houses N	UnGuided	Bucket-wise	Sync	mql$_9$	mql$_{15}$	quail	chimp
12	0.91	0.35	0.48	0.10	191.04	72.65	156.03
13	3.21	0.80	1.36	0.11	195.20	t/o	t/o
14	12.00	1.80	4.34	0.11	194.16		
15	49.36	4.51	10.20	0.14	192.91		
16	206.52	12.08	30.87	0.16	191.08		
17	t/o	34.00	139.63	0.18	190.48		
18		98.54	t/o	0.19	190.86		

Fig. 6. Comparison of the algorithm to other tools. CPU time in seconds for "all houses" from Smart Grid case study of [1].

We compare mql[1] [3], quail[2] [2] and chimp[3] [4] with our tool sharpPI. These tools calculate a precise Shannon entropy. We leave out tools which only returns an estimation of the information flow.

Figure 6 shows the runtime in seconds, measured on Intel Core(TM) i7 CPU 860 with 2.80 GHz and 8 GB RAM. The timeout is set to five minutes and the integer width of mql to 9 resp. 15 bits. sharpPI uses the Minisat. We select the timeout and the city block size N to a range, that shows differences between the tools.

Discussion. In direct comparison is mql the fastest tool with an a priori set integer width of 9 bits. With 15 bits, mql becomes slower with larger integer width, which determines mainly its run-time in this case study. quail and chimp separate magnitudes to the mql or sharpPI.

The Algorithm Bucket-wise is the fastest counting algorithm presented in this paper. We observe the reusing of a found models brings a performance gain (cf. Bucket-wise) and Sync to UnGuided, especially if it was found in the last call (Bucket-wise). One explanation could be the behavior of the decision stack in incremental SAT solver. An assumption is pushed as the first assignments on this stack. The decision and learned clauses are based on these assignments. If we use the same assumption in the next SAT solver call, the decision stack and all derived decisions are reusable.

4 Related Work

mql [3] uses Moped, a symbolic model checker, to calculate a boolean representation of a given program as an arithmetic decision diagram (ADD). The ADD encodes the function π, that maps the secret values to the observable values. Counting of the images and pre-images are reduced to operations on ADDs.

[1] https://sites.google.com/site/mopedqleak/, Access: 2016-07-15.
[2] https://project.inria.fr/quail/, Version: 2.0.
[3] http://www.cs.bham.ac.uk/research/projects/infotools/chimp/, Version: 2.1.

QUAIL [2] uses Markov Decision Procedure (MDP), that are built by depth-first search for the final states on the given program. The specification of secret and observable variables are fixed during execution and like our information flow model, the authors assumes completely defined start state. Finally, for the calculation of the entropy the MDP is striped down to discrete-time Markov chains (DTMC). CHIMP [4] builds directly an DTMC of the program in a similar fashion as QUAIL, but with a different information flow model, allowing partial assigned start state. MQL, QUAIL and CHIMP support probabilistic programs with their own input language.

5 Conclusion

We presented three different algorithms UNGUIDED, BUCKET-WISE and SYNC for the counting of images and pre-images of deterministic C programs encoded as CNF formulas, which utilizes the advantages of incremental SAT solvers. Algorithm BUCKET-WISE is by far fastest algorithm of our three introduced algorithms. In comparison with other tools, SHARPPI performs well against MQL, QUAIL and CHIMP for deterministic programs. We provide an implementation of all introduced algorithms in our tool SHARPPI.

References

1. Biondi, F., Legay, A., Quilbeuf, J.: Comparative analysis of leakage tools on scalable case studies. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 263–281. Springer, Heidelberg (2015)
2. Biondi, F., Legay, A., Traonouez, L.-M., Wąsowski, A.: QUAIL: a quantitative security analyzer for imperative code. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 702–707. Springer, Heidelberg (2013)
3. Chadha, R., Mathur, U., Schwoon, S.: Computing information flow using symbolic model-checking. In: Raman, V., Suresh, S.P., (eds.) 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS), vol. 29 of Leibniz International Proceedings in Informatics (LIPIcs), pp. 505–516, Dagstuhl, Germany, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2014)
4. Chothia, T., Kawamoto, Y., Novakovic, C., Parker, D.: Probabilistic point-to-point information leakage. In: Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013), pp. 193–205. IEEE Computer Society, June 2013
5. Klebanov, V., Manthey, N., Muise, C.: SAT-based analysis and quantification of information flow in programs. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 177–192. Springer, Heidelberg (2013)
6. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
7. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)