Precise Quantitative Information Flow Analysis— A Symbolic Approach

Vladimir Klebanov

Karlsruhe Institute of Technology, Germany

Abstract

Quantitative information flow analysis (QIF) is a portfolio of software security assessment techniques measuring the amount of confidential information leaked by a program to its public outputs. In this paper, we extend the scope of precise QIF for deterministic imperative programs where information flow can be described with linear integer arithmetic. We propose two novel QIF analyses that precisely measure both residual Shannon entropy and min-entropy of the secret and that feature improved tolerance to large leaks and large input domains.

For this purpose, we investigate the use of program specifications in QIF. We present criteria for specification admissibility and a program analysis that replaces exhaustive program exploration with symbolic execution, while incorporating user-supplied (but machine-checked) specifications. This kind of program analysis allows to trade automation for scalability, e.g., to programs with unbounded loops. Furthermore, we show how symbolic projection and counting, based in this instance on symbolic manipulation of polyhedra, avoid subsequent leak enumeration and enable precise QIF for programs with large leaks.

Keywords: Security, information flow, quantitative analysis, polyhedral model, symbolic model counting, program specifications

1. Introduction

Quantitative information flow analysis (QIF) is a portfolio of techniques for security assessment of software [1]. The research in QIF is motivated by the observation that it is not feasible to completely prevent information leaks (i.e., the flow of confidential information to public ports) in realistic systems. Instead, practical security analysis demands a measure of leaked information in order to decide if a leak is tolerable.

Information flow (alias leakage) in a program p (assumed known to the attacker) can be described by means of an equivalence relation $\approx_p^E$ showing which

Email address: klebanov@kit.edu (Vladimir Klebanov)

confidential inputs are indistinguishable by public program outputs. The relation $\approx_p^E$ is parametrized by the set E of experiments, i.e., public program inputs that an attacker chooses to enact. Information leaks correspond to equivalence classes of this indistinguishability relation. The more numerous and the smaller the classes, the more secret information is leaked by the program. A variety of information-theoretical security metrics can be used to summarize the situation and assess program security.

Our aim is to advance the state of the art in QIF for the class of deterministic imperative programs whose information flow can be described with linear arithmetic. We present two novel QIF analyses that precisely measure both number and size of equivalence classes and feature improved tolerance for the case that any of these values is large, as well as for large program input domains. The analyses are logic-based, i.e., they build on inference in a logical framework as opposed to, for instance, dynamic analysis (e.g., [2]) or type systems (e.g., [3]). Historically, our research has been inspired by [4], one of the pioneering works of logic-based QIF, but also by the preceding work on formalizing (qualitative) information leakage in a general-purpose program logic [5].

Logic-based QIF for deterministic imperative programs has been a fruitful research direction, bringing forward a multiplicity of analyses (e.g., [4, 6–11]) with a variety of underlying technologies. A consolidating observation we wish to make is that, despite the diversity, these analyses can be decomposed into four abstract components described declaratively. The components are:

1. static program analysis/verification condition generation
2. projection[1]
3. counting
4. calculations based on information theory.

It is worth noting that the first three components are, on this level of abstraction, not unique to QIF, and QIF can profit from research related to these components in other application areas.

Our analyses follow the same component structure, and we present an implementation of each component (resp. two implementations for component 1) that has not been used for QIF before. The bigger novelty of our proposal is in that all four components are symbolic. Rather than enumerate certain sets (e.g., the set of feasible program outputs), they manipulate potentially more compact *descriptions* of these sets. Enumeration bottlenecks are one of the reasons forcing existing approaches either to severely limit the input state space or to use approximation in at least one of the components, giving up soundness or precision. Symbolic reasoning enables us to treat programs that could not be treated before in a manner that is sound and precise.

[1] Projection (see, e.g.,[12]) is also known as image computation, range computation, (existential) quantifier elimination, or forgetting. In contrast to, e.g., counting, projection is often not emphasized in QIF literature, although it is at least as essential.

In detail, the main contributions of our work are:

- Component 1: a symbolic program analysis that replaces enumerative program exploration (e.g., bounded model checking) with an analysis incorporating user-supplied but machine-checked specifications such as loop invariants and procedure contracts. This enables QIF for programs with a large number of states/paths, and ultimately for programs that are beyond the current scope of automated program analysis. An important novel ingredient here is a criterion for specification admissibility for QIF. The implementation is based on the deductive verification system KeY [13].
- Components 2 and 3: projection and counting based on symbolic manipulation of polyhedral sets and relations. The implementation is based on the ISL/BARVINOK [14] framework. Together with the symbolic program analysis, these novel QIF component implementations avoid enumeration and enable precise measurement of equivalence class number *and* size beyond what was possible so far.[2]

We also make the following minor contributions:

- Component 4: a sketch of an approach for systematic symbolic computation of averaging security metrics (such as residual Shannon entropy) from a symbolic description of equivalence class sizes. The approach is based on Euler-Maclaurin summation.
- an extension of the QIF analyses above to non-uniform secret distributions and to non-terminating programs.

Furthermore, the component view and the chosen component implementations allow us to make another point regarding the structure of logic-based QIF analyses. These analyses populate so far two largely disconnected classes: those using the self-composition technique (examples are [4, 7]) and those operating on the program's transition relation (for example [8, 10]). While it is well-known that both approaches are sound, the differences in presentation and implementations so far obscured their commonalities. We make explicit the common structure, both theoretically (Proposition 2.3) and by implementing an instance of each class by a mere variation in component 1, while the other components remain essentially the same.

In the following, Section 2 provides formalizes information flow in programs, while Section 3 gives an overview of components that we combine for QIF.

[2] At this point it is instructive to point out the parallels and differences between our approach and [4]. Similarities are the overall component structure of the analysis, the use of self-composition in Component 1 (optional in our case), and the use of Barvinok's counting algorithm in Component 3. The difference is that [4] uses leak enumeration in all components. In particular, it combines the basic Barvinok's algorithm with equivalence class enumeration for counting. We use the more powerful symbolic variant of the algorithm for counting, and, altogether, non-enumerating implementations of all components.

Section 4 presents the actual QIF analyses. How to incorporate program specifications is shown in Section 5. Section 6 reports on experiments. Extensions are presented in Section 7. Section 8 surveys related work, and Section 9 concludes.

2. Background on Information Flow in Programs

2.1. Programs and states

We assume a deterministic, imperative programming language with bounded integer variables. For succinctness, we will denote several related program variables or terms as $\bar{\mathtt{v}}$, $\bar{t}$, etc. and assume that all operations happen componentwise. Unless noted otherwise, we assume for concreteness' sake that the integer data type covers the range $[-2^{31}, 2^{31} - 1]$.

A *program state* is a logical structure assigning values to program variables. We refer to the set of all possible states for a given program as S. Every syntactically valid program p describes a *transition relation* on program states $\rho_p \subseteq S \times S$. If the program p started in state s terminates in state s', then (and only then) $(s, s') \in \rho_p$. We call pairs of such initial and final states a *run* of p. The correspondence between p and ρ_p is fixed by the definition of the programming language, and we will show later in the paper how to compute ρ_p using verification technology.

We only consider deterministic programs. This means that all transition relations ρ_p are actually partial functions: for every initial state, there is at most one final state. We will use the functional or the relational notation as convenient. For modeling reasons, one may sometimes wish to restrict the set of initial states of a program to $S^{pre} \subset S$. Then $\rho_p \subseteq S^{pre} \times S$.

2.2. Formalizing Information Flow

The security lattice. To define information flow, we classify parts of program states according to a simple security lattice: a program variable can be marked either as a *secret* alias *high input* (initial value is confidential) or as a *public* alias *low input* (initial value is chosen by the attacker). Independent from that, a variable can be marked as a *public* alias *low output* (final value visible to the attacker). This labeling induces the respective projection functions $\cdot_{hi}$, $\cdot_{li}$ and $\cdot_{out}$ on states and sets of states. We assume, for every state s, that the high and low input components are disjoint ($s_{hi} \cap s_{li} = \emptyset$) and that together they completely determine every outgoing program run[3].

We will use the syntactical convention that the variables `h`, `l`, `o` are the high input, the low input, and the low output respectively. Furthermore, whenever

[3] In case that *every* variable in the program is either a high or a low input (i.e., $S = S_{hi} \times S_{li}$), the assumption of a deterministic programming language above suffices. If there are variables that are not part of the program input (e.g., local or auxiliary variables), they must be ruled out as sources of nondeterminism. A simple syntactical analysis can ensure that they have a fixed initial value or that every read from them is preceded by a write. The Java compiler, for instance, enforces the latter property for local variables.

we say "input" in the following, we may mean either an input variable or its initial value, which in its turn means the valuation provided by an initial state of an implied program run. An analogous situation holds for an output and its final value. The exact meaning should be clear from the context.

The attacker. The attacker model is as follows. For a single run of a program p with an initial state $s \in S^{pre}$ and the final state s', the attacker knows p, S^{pre}_{hi}, s_{li}, s'_{out}, and nothing else. The goal of the attacker is to learn something about s_{hi}. Beyond that, we assume that the attacker observes a *set* of runs, or *experiments*, in terminology of [4]. The secret input remains the same in all experiments, but the attacker gets to choose the low input, i.e., the low input component s_{li}, of each run's initial state. The amount of information leaked by the program (and thus the success of the attacker) depends on the number of experiments that the attacker can study. We refer to the totality of the low input state components chosen by the attacker as E. This set appears as a parameter in the QIF analysis presented below.

To treat low inputs, we consider a modified transition relation $\rho^e_p \subseteq S_{hi} \times S_{out}$, which, for a given $e \in S_{li}$, is obtained from ρ_p by fixing the initial low input state component to e and considering only the attacker-visible component of the final state, i.e., $(s_{hi}, s'_{out}) \in \rho^e_p$ iff $((e, s_{hi}), s') \in \rho_p$. For a set of experiments E, we generalize this construction to a natural join

$$\rho^E_p = \bowtie_{e \in E} \rho^e_p \ . \tag{1}$$

The relation $\rho^E_p \subseteq S_{hi} \times S^{|E|}_{out}$ relates each secret initial state component to an E-indexed tuple of public final state components.

Describing information leaks: the indistinguishability relation. Information flow in a program p can be identified with a particular equivalence relation $\approx^E_p$, the *indistinguishability relation* of p. This view has appeared at least as early as [15]; the development that we make below is closest in spirit to [16]. At the center is the well-known fact that every function induces an equivalence relation (its *equivalence kernel*) on its domain [17].

Definition 2.1 (Indistinguishability relation $\approx^E_p$)**.** The *indistinguishability relation* $\approx^E_p \subseteq S_{hi} \times S_{hi}$ is the equivalence kernel of the (functional) relation ρ^E_p, i.e.:

$$s_{1\,hi} \approx^E_p s_{2\,hi} \text{ iff for all } e \in E : \rho^e_p(s_{1\,hi}) = \rho^e_p(s_{2\,hi}) \ . \tag{2}$$

This equivalence relation relates any two secret inputs that are indistinguishable to the attacker in any experiment from E. The definition assumes that programs always terminate; we will consider the case that programs might not terminate in Section 7.2.

In the following sections, we will use a particular representation of $\approx^E_p$, that of an indistinguishability partition Π^E_p.

Definition 2.2 (Indistinguishability partition Π_p^E)**.** The $\approx_p^E$-induced *indistinguishability partition* Π_p^E is the quotient set $S_{hi}/\approx_p^E = \{C \mid C = \approx_p^E \circ s_{hi}$ for some $s_{hi} \in S_{hi}\}$.

Π_p^E partitions S_{hi} into equivalence classes of $\approx_p^E$. The partition is accompanied by the *natural projection* (Figure 1).

Definition 2.3 (Natural projection π)**.** The function $\pi\colon S_{hi} \to \Pi_p^E$, $s_{hi} \mapsto \approx_p^E \circ s_{hi}$ is the so-called *natural projection*, mapping each secret input to its $\approx_p^E$-equivalence class.

Furthermore, each member of Π_p^E is a subset of S_{hi} corresponding to some E-indexed tuple of low outputs of the program and containing exactly the secret inputs leading to this tuple of outputs under the experiment set E:

Proposition 2.1. $\Pi_p^E = \{C \mid C = (\rho_p^E)^{-1}(\bar{s_{out}})$ *for some* $\bar{s_{out}} \in \rho_p^E \circ S_{hi}\}$*, and furthermore,* $|\Pi_p^E| = |\mathit{ran}\,\rho_p^E|$.

Proof. This is a consequence of [17, Theorem 19], which states the existence of a unique bijective function $g\colon \Pi_p^E \to \mathit{ran}\,\rho_p^E$ with $\rho_p^E = g \circ \pi$. This function can be taken as homomorphic extension of ρ_p^E to equivalence classes of $\approx_p^E$, as ρ_p^E is by definition of $\approx_p^E$ constant on all elements of an equivalence class. □

Accordingly, one also speaks of members of Π_p^E (alias equivalence classes of $\approx_p^E$) as *preimages* of program outputs resp. output tuples.

Example 2.1. Consider the program `if(h==1) o=1 else o=0;`. If the set of experiments $E = \{17, 42\}$, then the indistinguishability partition $\Pi_p^E = \{\{-2^{31}, \ldots, 16, 18, \ldots, 41, 43, \ldots, 2^{31}-1\}, \{17\}, \{42\}\}$. The program leaks very little in each individual experiment, but expanding the set E will ultimately make the program leak the full secret.

Intuitively, an attacker can discern secret inputs from different classes in Π_p^E but not within one class. Secure programs have a coarse Π_p^E, while insecure a fine one. If $\approx_p^E$ is identity, then all classes are singleton sets (Π_p^E is very fine), and each output corresponds uniquely to a secret input: the attacker has perfect knowledge. Conversely, the coarsest indistinguishability relation $\approx_p^E = S_{hi} \times S_{hi}$ with only one class in Π_p^E means that the attacker learns nothing about the secret inputs by observing program outputs (a scenario known as "non-interference").

Proposition 2.2 ([17, Theorem 18])**.** *The equivalence kernel of* $\pi\colon S_{hi} \to \Pi_p^E$ *is the same as the one of* ρ_p^E*, i.e.,* $\approx_p^E$.

Since π talks about both states and sets of states, expressing it logically is complicated. Instead, it is easier to consider a slightly different function.

Definition 2.4 (Representative map π')**.** The *representative map* $\pi'\colon S_{hi} \to S_{hi}$ maps each secret input to a canonical representative of its $\approx_p^E$-equivalence class.

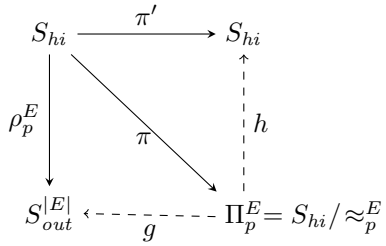


Figure 1: The indistinguishability relation $\approx_p^E$ is the equivalence kernel of the transition relation ρ_p^E, the natural projection π, or the representative map π'. The diagram commutes.

The function π' is not unique – it depends on how the canonical representative is chosen. Yet, for any such function, the following holds:

Proposition 2.3. *The equivalence kernel of* $\pi' \colon S_{hi} \to S_{hi}$ *is the same as the one of* ρ_p^E*, i.e.,* $\approx_p^E$*.*

Proof. We show that kernels of π and π' coincide. For this, we show that both functions have the same indistinguishability, i.e., $\pi(s_1) = \pi(s_2)$ iff $\pi'(s_1) = \pi'(s_2)$. The forward direction is obvious, in particular as π' is the canonical representative of its class. The reverse direction follows from the fact that $\pi'(s) \in \pi(s)$ (by definition of π') and that all equivalence classes are disjoint [17, Corollary of Theorem 18]. □

2.3. Security Metrics

For assessing security of a program p, it is customary to summarize the structure of $\approx_p^E$ resp. Π_p^E in a single real number signifying the information flow (leakage) of p. The leaked information is the difference between the attacker's initial uncertainty about the secret inputs and the residual uncertainty after observing the output of the program [18]. In the following, we concentrate on the residual uncertainty, which is obtained by applying one of the available security metrics to Π_p^E.

Some metrics, such as residual min-entropy [18], depend only on the number of classes in Π_p^E; some, such as minimal guessing entropy [19], are only concerned with the smallest class, and some, such as residual Shannon entropy [19], average over the class sizes. Our analyses can calculate both class number and sizes, and we use two popular metrics—conditional min-entropy and conditional Shannon entropy—as respective illustrations. It should be noted that different metrics have significantly different properties and are appropriate for different scenarios. None of the known metrics are superior to others *in all situations*. It may indeed be necessary to consider several metrics—or devise new metrics—in order to give dependable operational guarantees. We refer to [18] for a discussion.

To express the metrics, we associate the secret inputs with a random variable $\mathcal{H}$ ranging over S_{hi}, and the public outputs with a random variable $\mathcal{O}$ ranging over $S_{out}^{|E|}$. The program p and the set of experiments E restrict the values of $\mathcal{H}$ and $\mathcal{O}$ that can occur simultaneously. By default, we assume that $\mathcal{H}$ follows a uniform distribution, i.e., that all secret inputs are equally likely. We will consider the case of non-uniform distribution in Section 7.1.

Under these assumptions, we can compute [18, 19] the conditional Shannon entropy $H(\mathcal{H}|\mathcal{O})$ and the conditional min-entropy $H_\infty(\mathcal{H}|\mathcal{O})$ as follows:

$$H(\mathcal{H}|\mathcal{O}) = \frac{1}{|S_{hi}|} \sum_{C \in \Pi_p^E} |C| \log_2 |C| \qquad \text{and} \qquad H_\infty(\mathcal{H}|\mathcal{O}) = \log_2 \frac{|S_{hi}|}{|\Pi_p^E|} \tag{3}$$

The former is a lower bound in bit on the expected message length needed to communicate the remaining secret about $\mathcal{H}$ after observing $\mathcal{O}$. The latter is a measure in bit reflecting the probability of correctly determining $\mathcal{H}$ in a single guess after observing $\mathcal{O}$.

We note that if $|\Pi_p^E| = |S_{hi}|$ (i.e., $\approx_p^E$ is an identity relation), then $H(\mathcal{H}|\mathcal{O}) = H_\infty(\mathcal{H}|\mathcal{O}) = 0$ and the attacker has perfect knowledge. The case $|\Pi_p^E| = 1$ corresponds to $H(\mathcal{H}|\mathcal{O}) = H_\infty(\mathcal{H}|\mathcal{O}) = \log_2 |S_{hi}|$.

3. Components for QIF: Symbolic Computing with Programs, Sets, and Relations

3.1. Program Analysis: Verification Condition Generation

When we later describe how to transform programs into logical descriptions of their transition or indistinguishability relations, we will do so in terms of verification condition generation. In particular, we will use an abstract operator $vcg(\cdot)$ that takes a program-containing formula in the input language given below and returns an equivalent formula in first-order logic with arithmetic but without programs.

VCG Input Syntax. We use an input language for $vcg(\cdot)$ inspired by Dynamic Logic [20]. The language extends standard first-order logic with arithmetic with two predicate transformers. For every program p and every formula ϕ, $\langle p \rangle \phi$ ("diamond") and $[p]\phi$ ("box") are formulas. We note that the language is closed under subformula relation; in particular box and diamond operators can appear nested.

The diamond is a weakest precondition predicate transformer (also known as $wp(p, \phi)$). The diamond formula $\langle p \rangle \phi$ is true in a state $s \in S$, if the program p started in s terminates and the formula ϕ is true in the state $\rho_p(s)$ reached upon termination.

The box is a weakest liberal precondition predicate transformer (also known as $wlp(p, \phi)$). A box formula is a relaxation of the corresponding diamond formula in that termination is not required: $[p]\phi$ is true in $s \in S$, if either p does not terminate when started in s, or $\langle p \rangle \phi$ is true in s. The formula $\psi \to [p]\phi$ has the same intuitive meaning as the triple $\{\psi\}p\{\phi\}$ in Hoare Logic.

Terms and formulas without box or diamond operators have the meaning as usual in first-order logic. A formula is *logically valid* if it is true in every state.[4]

Implementing VCG. In general, numerous systems exist that offer suitable VCG functionality. In our experiments (cf. Section 6), we have used the KeY verification system for this purpose. The KeY system [13] is a deductive verification system (i.e., a theorem prover) for Java based on symbolic execution in Java Dynamic Logic. It includes the box and diamond operators as part of the input syntax, so the program logic formulas shown in this paper can be supplied to the system virtually verbatim. The approach, though, can be easily transferred to systems using, e.g., Hoare Logic.

VCG for unbounded loops. To deal with unbounded loops, the $vcg(\cdot)$ operator is typically parametrized by a user-supplied (or synthesized) loop invariant. For the purposes of precise QIF, special admissibility requirements apply to loop invariants and similar specifications. As this issue is transparent in the approaches that we describe below, we defer handling it to Section 5.

3.2. Manipulation of Polyhedral Sets and Relations

To manipulate the relations obtained during program analysis we propose to use the ISL/BARVINOK framework [21, 22]. The framework implements a portfolio of algorithms on polyhedral sets and relations. The ones relevant for our purposes are surveyed below. The included definitions are adapted from [23]. The advantage of the framework are its versatility and the symbolic nature of computation.

Definition 3.1 (Polyhedral sets and relations)**.** A (parametric) *polyhedral set* is a finite union of basic parametric sets $S = \bigcup_i S_i$, each of which can be represented using quasi-affine constraints

$$S_i = \lambda\bar{p}.\{\bar{x} \in \mathbb{Z}^d \mid \exists \bar{z} \in \mathbb{Z}^e : A\bar{x} + B\bar{p} + D\bar{z} \geqslant \bar{c}\}$$

where $\bar{p}$ is a parameter vector, and $A \in \mathbb{Z}^{m\times d}$, $B \in \mathbb{Z}^{m\times n}$, $D \in \mathbb{Z}^{m\times e}$, and $C \in \mathbb{Z}^m$, for some $m, n \geqslant 0$.

A (parametric) *polyhedral relation* is a finite union of basic parametric relations $R = \bigcup_i R_i$, each of which can be represented using quasi-affine constraints

$$R_i = \lambda\bar{p}.\{(\bar{x}_1, \bar{x}_2) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists \bar{z} \in \mathbb{Z}^e : A_1\bar{x}_1 + A_2\bar{x}_2 + B\bar{p} + D\bar{z} \geqslant \bar{c}\}$$

where $\bar{p}$ is a parameter vector, and $A_i \in \mathbb{Z}^{m\times d_i}$, $B \in \mathbb{Z}^{m\times n}$, $D \in \mathbb{Z}^{m\times e}$, and $C \in \mathbb{Z}^m$, for some $m, n \geqslant 0$.

The *parameter domain* of a parametric polyhedral set or relation is the non-parametric polyhedral set containing exactly those parameter values $\bar{p} \in \mathbb{Z}^n$, for which the parametric set or relation are not empty. A non-parametric set or relation is a special case with $n = 0$.

[4] Thus, there is implicit universal quantification over program variables.

On the concrete level, polyhedral sets and relations are described as comprehensions with constraints in first-order logic with linear arithmetic and converted to a disjoint union of basic sets/relations internally. The operations that ISL/BARVINOK provides are, among many others, the following.

Operation: Lexicographic optimization. The *lexmin* operator on parametric polyhedral sets computes the lexicographical minimum of the set as a function of the parameters—or detects that the set is empty. If R is a polyhedral relation, then $lexmin\, R = \big\{(\bar{a}, \bar{b}) \in R \mid \bar{b} = lexmin\{\bar{x} \mid (\bar{a}, \bar{x}) \in R\}\big\}$, i.e., only the smallest element remains in each domain element's image after lexicographical minimization of R. The technique behind *lexmin* is parametric integer programming (PIP) [24]. PIP's runtime is exponential in the worst case, though it is known to behave well on many inputs in practice. There is also a dual operator *lexmax*.

Operation: Projection. Projection means handling the existentially quantified variables in set and relation constraints. It is implemented as a cascade of a number of efficient quantifier elimination procedures for common special cases and lexicographical optimization [25]. The *ran* operator computes a relation's range (i.e., image). It is equivalent to projecting out (i.e., existentially quantifying) the domain of the relation.

Operation: Computing the inverse of a relation.

Operation: Computing cardinality. The $|\cdot|$ operator returns the cardinality of a parametric polyhedral set expressed as a piecewise quasi-polynomial in the parameters of the set.

Definition 3.2 (Quasi-polynomials)**.** A *quasi-polynomial* is a polynomial expression that may involve greatest integer parts (i.e., floors) of affine expressions of parameters and variables. A *piecewise quasi-polynomial* is a subdivision of a given parameter domain into disjoint polyhedral *chambers* with a quasi-polynomial associated to each chamber. The value of the piecewise quasi-polynomial at a given point is the value of the quasi-polynomial associated to the chamber that contains the point. Outside of the union of chambers, the value is assumed to be zero.

For relations, the cardinality operator computes not the number of tuples in the relation, but the number of elements in the image of each domain element: $|R| = \lambda p.|\{s \mid (p, s) \in R\}|$.

Example 3.1. The result of computing cardinality of the following relation

$$\big|\{(n, m) \mid \exists a\colon m = 3a \wedge 1 \leqslant m \leqslant n\}\big| = \lambda n. \begin{cases} \lfloor n/3 \rfloor & \text{if } n \geqslant 3 \\ 0 & \text{otherwise} \end{cases}$$

is a piecewise quasi-polynomial with only one chamber (though we show the implicit "otherwise" case for clarity here).

Counting of integer points in parametric polyhedral sets begins with *chamber decomposition*, i.e., splitting the parameter domain into regions where the set

has the same geometrical "shape". The number of chambers is polynomial in the size of the description of the set (in bit) when the dimension is fixed [14, Lemma 3]. In each chamber, Barvinok's counting algorithm [14, 26] based on a compact representation of the generating function of the set is applied. The algorithm is polynomial in the size of the set's description when the dimension is fixed [14, Proposition 2].

4. Two Novel QIF Analyses

In this section, we present two novel instances in the two popular classes of QIF analyses. We call the first class SpecGen as it uses the program analysis to compute a formula (i.e., a specification) describing the transition relation ρ_p^e. We call the second class SelfComp as it uses the program analysis to compute an explicit representation of $\approx_p^E$ using the technique of self-composition [5, 27, 28]. Both classes follow up with projection and counting.

The novelty of our instances is in the symbolic computation of both indistinguishability class number $|\Pi_p^E|$ and class size $\{|C| \mid C \in \Pi_p^E\}$. It is also satisfying to see that the formulation of the analyses (as actual input to the reasoner) is very close to the mathematical formulations of Section 2.2. Furthermore, the formulation makes explicit the very similar abstract structure of the two analysis classes. Nonetheless, there are also subtle differences between the two.

When implementing SpecGen in the polyhedral framework, the transition relation must be polyhedral, while SelfComp only requires that the indistinguishability relation is polyhedral, which is a much weaker requirement. In the latter case, the program may use complicated data structures (arrays, objects, etc.), as long as they are created from the inputs internally and the program analysis supports them. Of course, every finite relation is polyhedral, so we are primarily targeting relations that are "naturally" polyhedral.[5] Unfortunately, this property cannot be effectively determined from the program syntax. It only becomes apparent after the program analysis stage.

*4.1. Precise QIF from Transition Relations (*SpecGen*)*

This approach is based on specification generation. A verification condition generator is used to generate, for a given program p, a first-order logic formula Θ describing the same transition relation on states (which is then analyzed further).

[5] For efficiency reasons, we would not consider the relation $\{(x, y, z) \mid z = x \cdot y \wedge 0 \leqslant x, y < 100\}$ "naturally" polyhedral, even though the problematic multiplication of two variables *could* be replaced by a big disjunction.

4.1.1. Component 1: Program Analysis

Proposition 4.1. *A specification of a program p can be computed as the verification conditions of the formula:*

$$\Psi \equiv E(\bar{\mathtt{l}}) \wedge Pre \wedge \langle p \rangle \bar{\mathtt{o}} = \bar{o}' \tag{4}$$

where $\bar{o}'$ is a fresh constant serving as a place holder for program output. The predicate E describes the set of experiments. The optional precondition Pre over the high vocabulary of p allows restricting the set of initial states. The resulting specification $\Theta := vcg(\Psi)$ is a program-free constraint relating the output $\bar{o}'$ to the inputs $\bar{\mathtt{l}}$ and $\bar{\mathtt{h}}$.

Proof. By construction. If p contains unbounded loops, then specifications for these (loop invariants) need to be supplied to $vcg()$. This issue will be discussed in Section 5. Already here, we would like to note that the generated specification is the strongest one, unless the supplied invariants are not the strongest for their respective code pieces.

4.1.2. Components 2+3: Projection and Counting

Proposition 4.2. *Given $\Theta(\bar{\mathtt{l}}, \bar{\mathtt{h}}, \bar{o}')$ as the specification of p, a set of experiment values $\{\bar{e}_1, \bar{e}_2, \ldots, \bar{e}_k\}$ with $E(e_i)$ holding, the size of the equivalence classes of $\approx_p^E$ can be computed as a closed expression using the operations of Section 3.2 as follows:*

$$\{|C| \mid C \in \Pi_p^E\} = |(\rho_p^E)^{-1}| \ , \tag{5}$$

and the number of equivalence classes can be computed as

$$|\Pi_p^E| = |\, ran\ \rho_p^E| \ , \tag{6}$$

where ρ_p^E in this context is an abbreviation for

$$\{(\bar{\mathtt{h}}, (\bar{o}'_1, \ldots, \bar{o}'_k)) \mid \Theta(\bar{e}_1, \bar{\mathtt{h}}, \bar{o}'_1) \wedge \ldots \wedge \Theta(\bar{e}_k, \bar{\mathtt{h}}, \bar{o}'_k)\} \ . \tag{7}$$

Proof. It is easy to see that (7) indeed describes ρ_p^E (1), as the notation suggests. The claims (6) and (5) follow from Proposition 2.1. In context of (5), we recall that the cardinality operation on relations gives us a function mapping each element in the domain to the number of elements in its image. □

It is clear that SpecGen is not appropriate when the set of experiments E is large, since these need to be enumerated when constructing the description of ρ_p^E in (7). SelfComp does not have this limitation.

4.2. Precise QIF from Indistinguishability Relation (SelfComp)

4.2.1. Component 1: Program Analysis

This component employs a program analysis to compute a logical description of $\approx_p^E$. This is accomplished with self-composition [5, 27, 28], i.e., we employ two copies of the program $p(\bar{\mathtt{h}}, \bar{\mathtt{l}}, \bar{\mathtt{o}})$ with renamed variables: $p_1 := p(\bar{\mathtt{h}}_1, \bar{\mathtt{l}}_1, \bar{\mathtt{o}}_1)$ and

$p_2 := p(\bar{\mathtt{h}}_2, \bar{\mathtt{l}}_2, \bar{\mathtt{o}}_2)$. The goal is to determine a program-free formula $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$ such that

$$s_{1\,hi} \approx_p^E s_{2\,hi} \text{ iff } \Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \text{ is true in a state } (s_{1\,hi} \oplus s_{2\,hi}, s_{li}) \ ,$$

where s_{li} is some low state component and $s_{1\,hi} \oplus s_{2\,hi}$ is a high state component where the values of $\bar{\mathtt{h}}_1$ are the same as the values of $\bar{\mathtt{h}}$ in $s_{1\,hi}$ and the values of $\bar{\mathtt{h}}_2$ are the same as the values of $\bar{\mathtt{h}}$ in $s_{2\,hi}$.

Transcribing the characterization of indistinguishability (2) in program logic we obtain:

$$\Psi \equiv E(\bar{\mathtt{l}}_1) \wedge (\bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2) \wedge \mathit{Pre} \wedge \langle p_1 \rangle \langle p_2 \rangle (\bar{\mathtt{o}}_1 = \bar{\mathtt{o}}_2) \ . \tag{8}$$

As in SpecGen, the predicate E describes the set of experiments. The set of initial states can be restricted by including *Pre*, which in SelfComp is a (symmetric) precondition over the high vocabulary of p_1 and p_2. The described indistinguishability relation then only covers the high state components satisfying *Pre*.

The desired logical description of $\approx_p^E$ is immediately $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \equiv \mathit{vcg}(\Psi)$. Compared to SpecGen, SelfComp is less efficient in the program analysis stage, since the program in (8) has to be duplicated, which is not the case in (4).

4.2.2. Components 2+3: Projection and Counting

Proposition 4.3. *Using the operations of Section 3.2, the size of the equivalence classes of $\approx_p^E$, can be computed as a closed expression as follows:*

$$\{|C| \mid C \in \Pi_p^E\} = |(\mathit{lexmin} \approx_p^E)^{-1}| \ , \tag{9}$$

and the number of equivalence classes can be computed as

$$|\Pi_p^E| = |\, \mathit{ran}\ \mathit{lexmin} \approx_p^E \,| \ , \tag{10}$$

where $\approx_p^E$ in this context is an abbreviation for

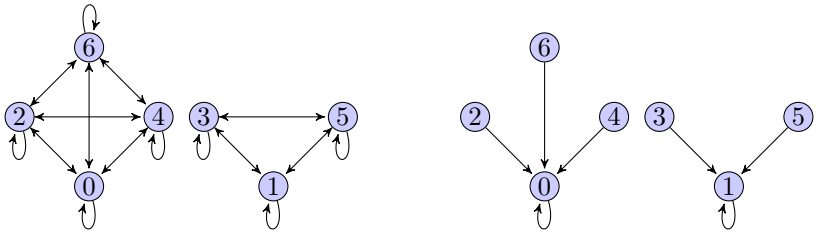
$$\{(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \mid \Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)\} \ . \tag{11}$$

Proof. We note that (11) is per construction a description of $\approx_p^E$.

We now consider the relation $\mathit{lexmin} \approx_p^E$. We claim that $\mathit{lexmin} \approx_p^E$ is nothing else than a particular choice of function π' from Figure 1.[6] By definition of *lexmin* (cf. Section 3.2), $\mathit{lexmin} \approx_p^E$ is the filtrate of $\approx_p^E$, such that each element in the relation's domain is no longer mapped to all elements in its equivalence class but merely to the unique lexicographically minimal element in that class. This minimal element serves as the representative of the class.

The claims (10) and (9) then follow from Propositions 2.1 and 2.3. In context of (9), we recall that the cardinality operation on relations gives us a function mapping each element in the domain to the number of elements in its image. □

[6] In this sense, the *lexmin* operator belongs to Component 1. It is instead included here due to the underlying technology it shares with Components 2 and 3.



(a) $\approx_p^E$ (b) $\pi' = (lexmin \approx_p^E)$

Figure 2: Computing the representative mapping π' of $\approx_p^E$ with lexicographical minimization

Example 4.1. Consider the program `o = h % 2;` with $0 \leqslant \mathtt{h} \leqslant 6$. The program leaks the parity of the secret input. The are no low inputs, and the indistinguishability relation $\approx_p^E$ can be described by the formula:

$$\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \equiv (\bar{\mathtt{h}}_1 \,\%\, 2 = \bar{\mathtt{h}}_2 \,\%\, 2) \wedge 0 \leqslant \bar{\mathtt{h}}_1 \leqslant 6 \wedge 0 \leqslant \bar{\mathtt{h}}_2 \leqslant 6 \ .$$

Figure 2a shows $\approx_p^E$ graphically. Please note that we have chosen a small modulo and small domains for illustration purposes only. Figure 2b shows the result of lexicographically minimizing $\approx_p^E$, which essentially computes a representative system for $\approx_p^E$. The representative is taken to be the smallest member in the class (0 and 1 for even and odd secrets, in this case).

The number of classes is the size of the range of the lexicographically minimized relation: $|\Pi_p^E| = |\, ran\ lexmin \approx_p^E |\, = 2$. To obtain the size of the classes, it is necessary to invert the relation $lexmin \approx_p^E$ and calculate the number of elements in the image of each point, which the cardinality operation on relations does: $\{|C| \mid C \in \Pi_p^E\} = |(lexmin \approx_p^E)^{-1}| = \{4 - h_2 \mid 0 \leqslant h_2 \leqslant 1\}$.

4.3. Component 4: Computing Security Metrics

In order to compute security metrics, the counting results must be combined with the appropriate formula(s), such as the ones presented in Section 2.3. The averaging class-size metrics (e.g., residual Shannon entropy) are more demanding in this regards, as they require computing sums of functions of quasi-polynomials. In the examples that we considered, it was always easy to compute residual Shannon entropy by applying elementary algebraic simplification laws (replacing repeated addition by multiplication, etc.). In the following, we sketch a general approach to efficient symbolic computation of averaging metrics using residual Shannon entropy as an example.

The approach is largely inspired by summation of quasi-polynomials over polyhedral domains, as studied in [29, 30] and implemented in ISL/BARVINOK. At the heart of the approach is the Euler-Maclaurin summation formula (EMF). The classical EMF [31] computes $\sum_{x=a}^{b} f(x)$, i.e., the sum of the function $f(x)$ over the one-dimensional polyhedral set $[a; b]$. EMF expresses the sum in terms

of an integral of f and a series of k terms based on higher-order derivatives of f, as well as a remainder term. The formula gives very exact approximations even without the remainder term and for low values of k. The advantage of the formula is that all terms above can be computed analytically. It is merely demanded that f has a continuous kth derivative.

Before applying EMF, a series of reductions [30] on the function describing the class size must be performed. The first reduction is from the piecewise quasi-polynomial to the quasi-polynomial case. This is achieved by splitting the summation domain into the (disjoint) chambers of the piecewise quasi-polynomial and combining the individual sums. We have not observed the number of chambers exceed a few in any of the examples we considered.

The second reduction is from the quasi-polynomial to the polynomial case. There are three different ways to achieve this goal. The first way is to "splinter" the summation domain. For example, the sum $\sum_{x=0}^{k} f(\lfloor \frac{x}{d} \rfloor, x)$ can be decomposed as follows

$$\sum_{x=0}^{k} f(\lfloor \frac{x}{d} \rfloor, x) = \sum_{t=0}^{\lfloor \frac{k}{d} \rfloor} f(t, dt) + \sum_{t=0}^{\lfloor \frac{k-1}{d} \rfloor} f(t, dt+1) + \ldots + \sum_{t=0}^{\lfloor \frac{k-d+1}{d} \rfloor} f(t, dt+d-1)$$

yet this may be inefficient when the "period" determined by the denominators of the quasi-polynomial (d in the above case) is large.

The second way is to introduce a new variable for each unique floor expression, as in

$$\sum_{x=0}^{k} f(\lfloor \frac{x}{d} \rfloor, x) = \sum_{\substack{0 \leqslant x \leqslant k \\ dt \leqslant x \leqslant dt+d-1}} f(t, x)$$

and then use EMF over higher-dimensional polyhedral sets. So far, EMF in higher dimensions have only been studied for polynomials [29, 30]. An extension to the case of general functions remains to be investigated.

The third way is by polynomial approximation. ISL/BARVINOK provides `lpoly` and `upoly` operations for point-wise under- and over-approximation respectively. The quality of approximations may vary, but the user obtains an error estimate. Since conditional Shannon entropy is monotonic in every point of the summation domain, the real value lies between the two approximations.

After the reductions, we are left with the sum of the function $f(x) = p(x) \log_2 p(x)$, where $p(x)$ is a polynomial. In the one-dimensional case, we know that the derivatives of f of order $k > 1$ are continuous except for a pole at every root of $p(x)$. While we also do know that $p(x) > 0$ in all integer points of the parameter domain (as the preimages are always non-empty), $p(x)$ could have non-integer roots. If x_0 is such a root, we propose to splinter the sum over the interval $[a; b]$ into two sums over intervals $[a; a_0]$ and $[a_0 + 1; b]$ with $a_0 < x_0 < a_0 + 1$. The roots of $p(x)$ can be efficiently determined with one of the various numerical root-finding algorithms, in particular as the summation domain is bounded, and the required precision is low.

5. Using Specifications for QIF

A powerful technique to reason about complex programs is to use specifications. Specifications are typically provided by the user (though sometimes also synthesized by a tool [32, 33]) and checked for correctness before they can be used. Two major uses of specifications are (i) replacing an unbounded loop by the loop invariant, and (ii) replacing multiple invocations of a procedure by the procedure contract.

For precise QIF, specifications must be sufficiently strong. Depending on the program, this may mean specifications that are either stronger (harder to find and prove) or weaker (easier to find and prove) in comparison to the ones typically used for verifying safety. In the following, we state two admissibility criteria for a specification: (i) that it is the strongest possible one for the given piece of code (and thus strong enough for precise QIF), and a more relaxed criterion (ii) that a specification is sufficiently strong for precise QIF w.r.t. the larger program where it is used.

5.1. Strength of Specifications

As described in Section 2.1, each program p induces a partial function ρ_p on states. The function $\rho_p(s)$ is defined exactly when p terminates if started in s. The value of $\rho_p(s)$ is then the final state of p.

Definition 5.1 (Specification of a program)**.** Every formula $I(\bar{\mathtt{x}}, \bar{x}')$ with program variables $\bar{\mathtt{x}}$ appearing in p and constants $\bar{x}'$ describes a relation on states R_I as follows: $(s_1, s_2) \in R_I$ iff $I(\bar{\mathtt{x}}, \bar{x}')$ holds in s_1 given that $\bar{x}'$ has the same value there as $\bar{\mathtt{x}}$ in s_2. $I(\bar{\mathtt{x}}, \bar{x}')$ is a specification of a program p, if $\rho_p \subseteq R_I$.

Note that specifications are allowed to overapproximate the behavior of the specified code. This is a standard definition of specifications. To show that $I(\bar{\mathtt{x}}, \bar{x}')$ is a specification of p, we have to show

$$\big(\langle p \rangle \bar{\mathtt{x}} = \bar{x}'\big) \to I(\bar{\mathtt{x}}, \bar{x}') \ . \tag{12}$$

If p contains a loop, showing (12) for some $I(\bar{\mathtt{x}}, \bar{x}')$ is as difficult as reasoning about p in the first place, so we demand specifications with more structure.

Proposition 5.1. *Let p be a loop of the form*[7] `while (`*cond*`) { `*body*` }`*. If $inv(\bar{\mathtt{x}}, \bar{x}')$ is a formula satisfying*

$$\bar{\mathtt{x}} = \bar{x}' \to inv(\bar{\mathtt{x}}, \bar{x}') \tag{13}$$

$$cond' \wedge inv(\bar{\mathtt{x}}, \bar{x}') \wedge (\bar{\mathtt{x}} = \bar{x}' \to [body]\bar{\mathtt{x}} = \bar{x}'') \to inv(\bar{\mathtt{x}}, \bar{x}'') \tag{14}$$

then $inv(\bar{\mathtt{x}}, \bar{x}') \wedge \neg cond'$ is a specification of p. Such a formula inv (called loop invariant*) always exists.*

[7] Here, *cond* is a side effect-free expression of type boolean. Every loop can be easily reduced to this normal form. Also, if there is a program part α preceding the loop, (13) becomes $\langle \alpha \rangle \big(\bar{\mathtt{x}} = \bar{x}' \to inv(\bar{\mathtt{x}}, \bar{x}')\big)$.

Proof. Consequence of [20, Theorem 3.2]. □

Condition (13) states that the invariant holds before the loop is entered, while condition (14) states that the invariant is preserved by the loop body. To show termination, loop specifications are typically accompanied by variants.

Definition 5.2 (Strongest specification)**.** $I(\bar{\mathtt{x}}, \bar{x}')$ is the strongest specification of a program p, if it is a specification of p, and in addition $\rho_p = R_I$.

The strongest specification of a program does not overapproximate its behavior, but captures it exactly. The strongest specification of a program (in the form compatible with Proposition 5.1) always exists [20].

Example 5.1. $I(\bar{\mathtt{x}}, \bar{x}') \equiv \bar{x}' > \bar{\mathtt{x}}$ is a specification of the program `x=x+1;` (assuming unbounded integers), while $I(\bar{\mathtt{x}}, \bar{x}') \equiv \bar{x}' = \bar{\mathtt{x}} + 1$ is its strongest specification.

5.2. Admissibility of Specification for Precise QIF

Replacing a program p by a specifications I is problematic for the purposes of precise QIF whenever the specification overapproximates program behavior. Depending on the chosen metric, unguarded use of specifications may cause either loss of precision or unsoundness. In the following we state two QIF-admissibility criteria for specifications, while a discussion of what happens when these criteria are not fulfilled follows after that.

Proposition 5.2. *A specification I of p satisfying*

$$\forall \bar{u}, \bar{v}, \bar{w}.\, I(\bar{u}, \bar{v}) \wedge I(\bar{u}, \bar{w}) \rightarrow \bar{v} = \bar{w} \tag{15}$$

is exactly the strongest specification. It is admissible for precise QIF.

Proof. Immediately by Definitions 5.1 and 5.2, since (15) means that R_I relates every state in its domain to at most one state. □

An obvious relaxation of (15) is to limit the low inputs to the values in E, if the specification is used at the beginning of a program. Furthermore, whenever only a part of the program is replaced by a specification, it is actually sufficient to use a specification that does not overapproximate behavior w.r.t. the *whole* program. At the same time, it is sufficient that the specification is strong enough to precisely determine merely the *public* behavior of the whole program. For example, the loop invariant *true* is in this sense sufficiently strong for the program `while (l>h1) {l--; h2--;} l=0;`. It is admissible to omit `l` from the invariant as `l` is always erased after the loop, and it is admissible to omit `h1` and `h2` from the invariant as they are neither visible to the attacker nor used to control public behavior after the loop. This notion of admissibility is formalized in the following.

Proposition 5.3. *Let the program p have the form $\alpha;\beta$ and I be a specification of α. Then, I is* sufficiently strong *w.r.t. p, if*

$$\forall \bar{u}, \bar{v}, \bar{w}, \bar{y}, \bar{z}.\, \big(I(\bar{u}, \bar{v}) \wedge I(\bar{u}, \bar{w}) \wedge ((\bar{\mathtt{l}}, \bar{\mathtt{h}}) = \bar{v} \to \langle\beta\rangle\bar{\mathtt{o}} = \bar{y}) \wedge ((\bar{\mathtt{l}}, \bar{\mathtt{h}}) = \bar{w} \to \langle\beta\rangle\bar{\mathtt{o}} = \bar{z})\big) \to \bar{y} = \bar{z} \quad (16)$$

In this case, I is admissible for precise QIF.

Proof. By (16) $\rho_\beta \circ R_I$ is a partial function with $\rho_\alpha \subseteq R_I$. Thus $\rho_\beta \circ R_I = \rho_{\alpha;\beta}$. □

5.3. What Happens when an Inadmissible Specification is Used

It is natural to ask what happens when a part of the program is replaced by an inadmissible specification. This question is studied in detail in [6], though not in the context of specifications but of approximative QIF. There, abstract interpretation—an automatic approximative program analysis that has the same general effect as using weak specifications—is used to derive leakage bounds.

Using an inadmissible specification increases the number of possible outputs (reachable final states), and thus $|\Pi_p^E|$. Conditional min-entropy (3) is anti-monotonic in $|\Pi_p^E|$, yielding a (conservative) lower bound on the residual uncertainty of the secret.

On the other hand, using an inadmissible specification also increases the preimage size as the preimages are no longer disjoint. For the conditional Shannon entropy (3), this implies an *upper* bound on the residual uncertainty of the secret. In other words, the analysis is unsound. It is possible to derive a lower bound for the residual Shannon entropy, but for this an under-approximation of the class sizes is needed. [6] uses, for instance, concolic testing—a program analysis that is exact but only considers a selected set of program paths—to generate such under-approximations.

5.4. Comparison with Direct Use of Abstract Interpretation

Specification synthesis is often based on abstract interpretation. On the other hand, QIF approaches like [6] apply a different abstract interpretation flavor to synthesize an approximation of the set of reachable final states of the program. In general, we advocate pragmatism in choosing the tool that works best for the given application scenario (e.g., w.r.t. performance, language coverage, expertise available on site, etc.). Yet, the following advantages to writing or synthesizing specifications as opposed to synthesizing the set of reachable states directly are worth mentioning.

(i) Admissibility of specifications can be proved. This provides a guarantee that no precision has been lost and that, e.g., the measured min-entropy leak is indeed real. (ii) If needed, specifications can be manually strengthened by domain experts (and then machine-checked) to achieve the necessary precision.

Table 1: Variations of the sum benchmark (projection and counting)

# summands	each (bit)	$\lvert\Pi_p^E\rvert$	time	$\{\lvert C\rvert \mid C \in \Pi_p^E\}$	time (m:s)
3	32	2^{32}	0	2^{64} each	0
10	1	11	0	between 1–252	0:21
10	32	2^{32}	0	$2^{288} \approx 4.9 \times 10^{86}$ ea.	4:22
32	1	33	0	t/o	t/o

Intel Core i7 860 2.80GHz CPU, time 0 = <50ms, t/o=time-out at 1h

(iii) Specifications can be used to precisely compute security metrics that involve class sizes (e.g., residual Shannon entropy). (iv) Specifications can be shared with many other verification tools for proving functional correctness of implementations.

6. Experiments

We evaluated our analyses on the 11 synthetic benchmarks collected in [10]. These benchmarks are quite diverse but often not challenging enough for our analyses, so we also considered more difficult versions. We describe selected benchmarks in detail to demonstrate interesting properties of our analyses.

Unless noted otherwise, all experiments were carried out on a system with an Intel Core i7 860 2.80GHz CPU. For program analysis we used the KeY system v1.6 [13, 34]. For projection and counting we used the `iscc` interactive shell of ISL/BARVINOK v0.35 [21, 22]. Both tools are publicly available. We did not implement any gluing code between KeY and ISL/BARVINOK, but since both systems use virtually identical formula syntax, it was easy to cut and paste between the two.

Projection and counting when determining $\lvert\Pi_p^E\rvert$ were instantaneous except in one case ("binary search", discussed below). We could also instantaneously determine class size (something that [10] is not concerned with) in base versions of all but the "population count" and "mix and duplicate" benchmarks. Together with the "illustrative example", these are the benchmarks that are not naturally polyhedral. We reencoded them manually on the individual bit level.

Sum of three numbers. The goal of this benchmark is to determine how much information is leaked by publishing the sum of three secret bounded integers: `o = h1 + h2 + h3;`. This example has appeared in [4, 7, 10] with the limitation $0 \leqslant \mathtt{h}_i \leqslant 9$. We establish precise class number and sizes in more challenging variations of this benchmark (Table 1).

We note that between row 2 and 3 the runtime increased roughly 13-fold, while the size of the secret increased by a factor of 2^{310}. The data in the table supports the theoretical result that the runtime of polyhedral manipulation is dictated by the shape of the relation and less so by the size of the domain (which is in general not true for, e.g., SAT/#SAT-based methods). The last table row is essentially the "population count" benchmark.

Electronic wallet. `o = 0; while (h >= l) { h = h - l; o = o + 1; }` is a program that receives a secret input `h` with the balance of a bank account and debits a fixed amount `l` from this account until the balance is insufficient for another debit transaction. The number of successful debit transactions (stored in `o` upon termination) reveals partial information about the initial balance of the account.

The standard version of this benchmark assumes $0 \leqslant \mathtt{h} < 20$. In contrast, we merely assume $0 \leqslant \mathtt{h} \leqslant 2147483647$. Like others, we do assume a single concrete experiment $\mathtt{l} = 5$. A concrete value is necessary to satisfy the polyhedral relation requirement. The loop invariant is easy to find and check for admissibility: $\left(\mathtt{h}_i = h'_i + 5\,o'_i\right) \wedge 0 \leqslant h'_i \leqslant 2147483647 \wedge \Lambda_i^{\textsc{SelfComp}}$. The primed and unprimed symbols are as in Definition 5.1.

When using SelfComp, the appropriately indexed invariant instance ($i = 1, 2$) must be used for each loop copy. The subformula $\Lambda_i^{\textsc{SelfComp}}$ must encode the fact that each loop copy does not affect the other's variables (i.e., it is of the form $v'_{3-i} = \mathtt{v}_{3-i}$ for each program variable `v`). This formula can be generated automatically, or eliminated altogether if the verification condition generator supports framed invariants (KeY, for instance, does). When using SpecGen, this formula as well as the variable indices are not used. The rest of the analysis is straightforward.

The electronic wallet is the only example with non-negligible program analysis running time reported in [4]. It took the model checker 24 seconds to compute $\approx_p^E$ for $0 \leqslant \mathtt{h} < 20$ (on unidentified hardware). We have repeated the experiment with KeY in exhaustive mode, treating the loop purely by unwinding. On a mobile system with a 1.60GHz Intel Core2 Duo CPU, computing $\approx_p^E$ took around 3.5 seconds with KeY. Computing ρ_p^E took around 0.5 seconds. We conjecture that the deductive analysis was faster, as it can compute the indistinguishability relation completely in one symbolic execution run. A model checker, in contrast, explores the full program repeatedly as the indistinguishability relation is refined with each new leak. In any case, unwinding is not a feasible option in the unrestricted version of this benchmark.

Binary search. This benchmark leaks `b` most significant bits of `h` (which is assumed to be an unsigned integer) to `o`: `o = 0; for (i = 0; i < b; i++) { if (h >= o + 2^(31-i)) o += 2^(31-i);}`. The base version assumes $\mathtt{b} = 16$, and following [10] we initially attempted to unwind the loop. Since all decisions are independent, the program has $2^{\mathtt{b}}$ paths.

Due to this fact, we could not compute ρ_p^E in KeY by unwinding, as KeY is not optimized for a high number of paths. Feeding a more efficient manual encoding of ρ_p^E to SpecGen, projection took just under two seconds, while counting timed out. Inspection showed that the image of ρ_p^E computed by projection was actually a *point enumeration* of the 65536 outputs—a case for which isl/barvinok is not optimized. In [10], approximative projection computation takes about 6.4 seconds on unidentified hardware.

The reason for the pathological projection result in our case is the combination of loop unwinding with isl/barvinok's disjoint basic relation decomposi-

tion. Replacing unwinding with a compact description of loop behavior—loop invariant—made the following analysis instantaneous for any `b`. With unwinding, the runtime increases steeply in this parameter, as [10] reports.

7. Extensions

7.1. Non-uniform secret distributions

Section 2.3 assumes that the secret inputs follow a uniform distribution. For the case of a non-uniform distribution, [35] gives a reduction to the uniform case. The reduction is based on a "pre-processing" program, which transforms the uniform distribution into the desired one. The approach is agnostic of the particular QIF analysis and is thus compatible with our proposal. At the same time, we show how non-uniform distributions can be naturally integrated with symbolic model counting in an alternative way, based on a weight function.

We model the non-uniform secret input distribution by means of the function $w\colon S_{hi} \to \mathbb{N}_0$. The function assigns each secret input (i.e., high state component) s_{hi} a non-negative integer weight $w(s_{hi})$ denoting its frequency of occurrence in the input distribution. The weight function can be seen as an "inverse" of the pre-processing program of [35]. The program describes which elements of the fictitious uniform distribution are conflated in reality, while the function gives the multiplicity of each real input in the fictitious uniform distribution. In a given application, one of the approaches may be more natural. The weight function, for instance, makes it easier to specify that certain secret inputs never appear.

Proposition 7.1. *For a non-uniform secret input distribution given by a weight function w and the indistinguishability partition Π_p^E as in Proposition 2.1:*

$$H(\mathcal{H}|\mathcal{O}) = \frac{1}{|S_{hi}|_w}\left(\sum_{C\in\Pi_p^E} |C|_w \log_2 |C|_w - \sum_{s_{hi}\in S_{hi}} w(s_{hi}) \log_2 w(s_{hi})\right) \tag{17}$$

$$H_\infty(\mathcal{H}|\mathcal{O}) = \log_2 \frac{|S_{hi}|_w}{\sum_{C\in\Pi_p^E} \max_{s_{hi}\in C} w(s_{hi})} \ , \tag{18}$$

where $|C|_w = \sum_{c\in C} w(c)$ is the weighted sum over all elements in C.

Proof. Proof of (17) proceeds along the lines of the proof of Theorem 1 in [35], while (18) follows from the derivation $H_\infty(\mathcal{H}|\mathcal{O}) = \log_2 \frac{1}{V(\mathcal{H}|\mathcal{O})}$ with the vulnerability $V(\mathcal{H}|\mathcal{O}) = \sum_{C\in\Pi_p^E} \max_{s_{hi}\in C} P[\mathcal{H} = s_{hi}]$ as shown in [18, p. 297]. □

The weighted sum operation $|\cdot|_w$ over polyhedral sets is an extension of the basic cardinality operation $|\cdot|$ and is directly available in the ISL/BARVINOK framework. It accepts a weight function given as a piecewise quasi-polynomial and computes the weighted sum over its domain. The result is as well a piecewise quasi-polynomial.

The case of the conditional min-entropy is more involved. We have to assume that w is a polyhedral relation—not a quasi-polynomial. Then, the denominator in (18) can be computed as $|\mathit{lexmax}\ \ w \circ ((\mathit{lexmin} \approx_p^E)^{-1})|_{\lambda x.x}$, where $\lambda x.x$ is a quasi-polynomial analog of an identity function. The term $\mathit{lexmin} \approx_p^E$ encodes the representative mapping π' as in Section 4.2.

7.2. Treating Leaks by Termination

We can extend the SelfComp characterization of indistinguishability (8) to measure leaks via termination (as done for proving insecurity in [5]):

$$E(\bar{\mathtt{l}}_1) \wedge (\bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2) \wedge \mathit{Pre} \wedge \left(\langle p_1 \rangle \langle p_2 \rangle (\bar{\mathtt{o}}_1 = \bar{\mathtt{o}}_2) \bigvee \left([p_1]\mathit{false} \wedge [p_2]\mathit{false} \right) \right) \ . \quad (19)$$

This way, we demand that either any two runs of the program do not terminate ($[p_i]\mathit{false}$ holds) or that their low outputs are indistinguishable.

The rest of SelfComp (i.e., projection and counting) can remain unchanged. It is only necessary that the program analysis supports reasoning about non-terminating programs (like, e.g., most deductive verification systems) and is applied in a way that is admissible for QIF in the sense of Section 5.

8. Related Work

There is a large body of work dedicated to both defining and demonstrating absence of information flow in programs. We refer to [36] for a survey.

A seminal work reasoning about information flow in program logic is [5], showing different approaches to formalize and prove both program security (absence of leaks) and insecurity (presence of leaks) in a general-purpose program logic. The self-composition technique was first presented in a workshop version of [5] and received further theoretical treatment and its name in [27]; it was also studied from the point of view of verification in [28].

A sound theory of information-theoretic quantification of information flow in imperative programs was originally developed in a series of works culminating in [3].

The theoretical hardness of QIF (under popular complexity-theoretical assumptions) has been shown by Terauchi et al. in [37]. As with other hard problems (e.g., SAT or SMT), such results merely show that hard instances can be constructed. They do not preclude the existence of efficient and practically relevant analyses for individual instances or subclasses of the problem.

A survey of QIF models and techniques is available in [1] and in [3]. In the following, we concentrate on QIF analyses for imperative programs, grouping them by the major QIF components.

In a SpecGen-style analysis, the transition relation resp. its approximation can be computed using bounded program unwinding (e.g., in model checker CBMC) [8, 11] or unwinding along selected program paths only (in [6] using a concolic testing tool, and in [9] using a binary code analyzer). In [10], the relations for the benchmarks are created manually. As the only non-enumerative

program analysis, [6] combines approximate transition relation computation and projection as abstract interpretation of a program.

SelfComp-style analyses, using self-composition to compute the indistinguishability relation, are not as common. [4] uses an iterative procedure that successively refines $\approx_p^E$ with counterexamples to indistinguishability generated by the model checker ARMC. [7] uses self-composition in a bounded model checker, but essentially only computes the transition relation. [38] composes n copies of the program to check whether $|\Pi_p^E| < n$, i.e., establish an upper bound on the leak. In the last case, QIF is formulated as a pure model checking problem.

Projection resp. its approximation can be computed using propositional model enumeration with a SAT solver [7, 8], model enumeration with the Omega Calculator (a tool for reasoning in quantified linear arithmetic) [4], binary search for models in a model checker [11], refinement with entailment queries along "2-bit patterns" [10], refinement with SMT entailment queries along intervals [9], and compilation of propositional formulas to d-DNNF [8].

Precise resp. approximative counting can be performed using basic (i.e., non-parametric) Barvinok's algorithm implemented in the LattE framework [4, 6], point enumeration (typically part of enumerative projection) [7, 8, 11], #SAT (counting solutions of propositional formulas [39]) [7, 8, 10], and in [9]: probabilistic approximate #SAT, sampling, and sum of interval sizes.

In certain applications, it may be also appropriate to design dedicated counting procedures. For instance, a procedure for counting concretizations of abstract cache-states is described in [40], where the authors are concerned with quantifying information flow in cache attacks.

We would like to note that enumerative techniques have both advantages and disadvantages. On the plus side, the procedures used for generating a single model are typically simpler and often extensively optimized. Furthermore, enumeration can be interrupted at any time, trading off computation time for approximation quality. On the minus side, enumeration is only feasible up to a certain number of models: enumerating 2^{32} models at 1 000 models per second requires about fifty days. An interesting hybrid between symbolic reasoning and enumeration are the refinement methods, where a whole family of models (e.g., an interval) are included or ruled out at a time.

Among the above approaches, only [4, 6–8] are concerned with indistinguishability class sizes. [6] proposes efficient probabilistic derivation of residual Shannon entropy using randomized sampling. The result is probably correct up to the user-chosen confidence level.

We are also aware of the following QIF analyses that do not immediately fit the component model. An extension of a security type system for an imperative language (including loops) with information flow bound calculation is proposed in [3]. An extension of a dynamic bitwise taint analysis for C programs is proposed in [2]. The latter technique has been applied to large programs used in practice. On the other hand, it only measures leakage along one or a few selected program paths, leaving it to the user to supply "representative" inputs. Another QIF tool based on dynamic analysis is reported in [41]. The tool auto-

matically derives bounds of information leakage in terms of mutual information and capacity from trial runs of the system, which is treated as a black box.

[42] extends the entropy-based model of information flow by incorporating the attacker's beliefs about the secret distribution (which may also be wrong). A mechanization of the belief-based model is not reported.

The interplay of specifications and implementations for non-interference is studied in [43]. The defined "Shadow Semantics" enables secure refinement of programs by demanding that nondeterminism is preserved at critical points. We, on the other hand, are concerned with preserving determinism.

9. Conclusion

We have extend the scope of precise QIF using symbolic polyhedral reasoning and a deductive program analysis that can use program specifications. The combination of symbolic components avoids the enumeration bottleneck and enables precise analysis for programs where it has not been possible before. The analyses we propose measure both number and size of indistinguishability classes with an improved tolerance to large values of both, as well as the size of the input domain.

In particular, program specifications enable precise QIF for programs with a large number of paths. While such analysis requires user input and takes time, this time is dependent on program complexity and not on the number of loop iterations or the size of involved data domains. The effort can be further amortized when QIF is combined with functional verification.

We have demonstrated that a deductive verification system and a polyhedral framework are a good platform for implementing a QIF analysis, and doing so in a satisfyingly direct fashion. Correctness of the analysis follows almost immediately from basic relational algebra and the soundness of the used reasoning tools. As a side result, we could make very clear the structural similarities between QIF analyses that utilize self-composition and those that do not.

Acknowledgment. This work was supported by the German National Science Foundation (DFG) under the priority programme 1496 "Reliably Secure Software Systems – RS3." The author would like to thank Bernhard Beckert for fruitful discussions, Sven Verdoolaege for his invaluable help with the ISL/BARVINOK framework, and the anonymous reviewers for suggestions on improving the presentation of this work.

References

[1] C. Mu, Quantitative information flow for security: a survey, Tech. Rep. TR-08-06, Department of Computer Science, King's College London, updated 2010, available at `http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-08-06.pdf` (2008).

[2] S. McCamant, M. D. Ernst, Quantitative information flow as network flow capacity, in: Proceedings, ACM SIGPLAN conference on Programming language design and implementation (PLDI), ACM, 2008, pp. 193–205.

[3] D. Clark, S. Hunt, P. Malacaria, A static analysis for quantifying information flow in a simple imperative language, Journal of Computer Security 15 (3) (2007) 321–371.

[4] M. Backes, B. Köpf, A. Rybalchenko, Automatic discovery and quantification of information leaks, in: Proceedings, 30th IEEE Symposium on Security and Privacy (S&P 2009), IEEE Computer Society, 2009, pp. 141–153.

[5] Á. Darvas, R. Hähnle, D. Sands, A theorem proving approach to analysis of secure information flow, in: D. Hutter, M. Ullmann (Eds.), Proceedings, Security in Pervasive Computing, Vol. 3450 of LNCS, Springer, 2005, pp. 193–209.

[6] B. Köpf, A. Rybalchenko, Approximation and randomization for quantitative information-flow analysis, in: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium, CSF '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 3–14.

[7] J. Heusser, P. Malacaria, Applied quantitative information flow and statistical databases, in: Proceedings of the 6th international conference on Formal Aspects in Security and Trust, FAST'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 96–110.

[8] V. Klebanov, N. Manthey, C. Muise, SAT-based analysis and quantification of information flow in programs, in: Proceedings, International Conference on Quantitative Evaluation of Systems, Vol. 8054 of LNCS, Springer, 2013, pp. 156–171, to appear.

[9] J. Newsome, S. McCamant, D. Song, Measuring channel capacity to distinguish undue influence, in: PLAS 2009, ACM, New York, NY, USA, 2009, pp. 73–85.

[10] Z. Meng, G. Smith, Calculating bounds on information leakage using two-bit patterns, in: PLAS 2011, ACM, 2011, pp. 1–12.

[11] Q.-S. Phan, P. Malacaria, O. Tkachuk, C. S. Păsăreanu, Symbolic quantitative information flow, in: P. Mehlitz, N. Rungta, W. Visser (Eds.), Proceedings, Java Pathfinder Workshop, 2012, pp. 1–5.

[12] C. Wernhard, Tableaux for projection computation and knowledge compilation, in: TABLEAUX 2009, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 325–340.

[13] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software: The KeY Approach, Vol. 4334 of LNCS, Springer, 2007.

[14] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, M. Bruynooghe, Counting integer points in parametric polytopes using Barvinok's rational functions, Algorithmica 48 (1) (2007) 37–66.

[15] J. Landauer, T. Redmond, A lattice of information, in: Computer Security Foundations Workshop VI, 1993. Proceedings, 1993, pp. 65–70. doi:10.1109/CSFW.1993.246638.

[16] D. Clark, S. Hunt, P. Malacaria, Quantitative information flow, relations and polymorphic types, J. Log. and Comput. 15 (2) (2005) 181–199.

[17] G. Birkhoff, S. MacLane, Algebra, AMS Chelsea Publishing, 1999.

[18] G. Smith, On the foundations of quantitative information flow, in: Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures, FOSSACS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 288–302.

[19] B. Köpf, D. Basin, An information-theoretic model for adaptive side-channel attacks, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, ACM, New York, NY, USA, 2007, pp. 286–296.

[20] D. Harel, First-Order Dynamic Logic, Springer, 1979.

[21] S. Verdoolaege, Counting affine calculator and applications, in: Proceedings, First International Workshop on Polyhedral Compilation Techniques, IMPACT'11, 2011.

[22] S. Verdoolaege, The BARVINOK library, Website at www.freecode.com/projects/barvinok.

[23] S. Verdoolaege, Polyhedral process networks, in: S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, J. Takala (Eds.), Handbook of Signal Processing Systems, Springer US, 2010, pp. 931–965.

[24] P. Feautrier, Parametric integer programming, RAIRO Recherche Opérationnelle 22 (3) (1988) 243–268.

[25] S. Verdoolaege, K. Beyls, M. Bruynooghe, F. Catthoor, Experiences with enumeration of integer projections of parametric polytopes, in: Proceedings of the 14th international conference on Compiler Construction, CC'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 91–105.

[26] A. I. Barvinok, A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed, Math. Oper. Res. 19 (1994) 769–779.

[27] G. Barthe, P. R. D'Argenio, T. Rezk, Secure information flow by self-composition, in: 17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA, IEEE Computer Society, 2004, pp. 100–114.

[28] T. Terauchi, A. Aiken, Secure information flow as a safety problem, in: C. Hankin, I. Siveroni (Eds.), Proceedings, Symposium on Static Analysis, Vol. 3672 of LNCS, Springer, 2005, pp. 352–367.

[29] N. Berline, M. Vergne, Local Euler-Maclaurin formula for polytopes, Mosc. Math. J. 7 (3) (2007) 355–386.

[30] S. Verdoolaege, M. Bruynooghe, Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison, in: M. Beck, T. Stoll (Eds.), The 2008 International Conference on Information Theory and Statistical Learning,, 2008, pp. 60–66.

[31] V. Lampret, The Euler-Maclaurin and Taylor formulas: Twin, elementary derivations, Math. Mag. 74 (2) (2001) 109–122.

[32] B. Weiß, Predicate abstraction in a program logic calculus, in: Proceedings of the 7th International Conference on Integrated Formal Methods, IFM '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 136–150.

[33] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, C. M. Wintersteiger, Loop summarization using state and transition invariants, Form. Methods Syst. Des. 42 (3) (2013) 221–261.

[34] The KeY tool, Website at `www.key-project.org`.

[35] M. Backes, M. Berg, B. Köpf, Non-uniform distributions in quantitative information-flow, in: B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, D. S. Wong (Eds.), Proceedings, 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS), ACM, 2011, pp. 367–375.

[36] A. Sabelfeld, A. C. Myers, Language-based information-flow security, IEEE J.Sel. A. Commun. 21 (1) (2006) 5–19.

[37] H. Yasuoka, T. Terauchi, Quantitative information flow – verification hardness and possibilities, in: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium, CSF '10, IEEE Computer Society, 2010, pp. 15–27.

[38] J. Heusser, P. Malacaria, Quantifying information leaks in software, in: Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10, ACM, 2010, pp. 261–269.

[39] C. P. Gomes, A. Sabharwal, B. Selman, Model counting, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, Vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009, pp. 633–654.

[40] B. Köpf, L. Mauborgne, M. Ochoa, Automatic quantification of cache side-channels, in: P. Madhusudan, S. A. Seshia (Eds.), Proceedings, Computer Aided Verification (CAV), Vol. 7358 of LNCS, Springer, 2012, pp. 564–580.

[41] K. Chatzikokolakis, T. Chothia, A. Guha, Statistical measurement of information leakage, in: Proceedings, international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer-Verlag, 2010, pp. 390–404.

[42] M. R. Clarkson, A. C. Myers, F. B. Schneider, Quantifying information flow with beliefs, J. Comput. Secur. 17 (5) (2009) 655–701.

[43] C. Morgan, Compositional noninterference from first principles, Formal Asp. Comput. 24 (1) (2012) 3–26.