What's the Over/Under? Probabilistic Bounds on Information Leakage

Ian Sweet[1], José Manuel Calderón Trilla[2], Chad Scherrer[2], Michael Hicks[1], and Stephen Magill[2]

[1]University of Maryland and [2]Galois Inc.

Abstract. Quantitative information flow (QIF) is concerned with measuring how much of a secret is leaked to an adversary who observes the result of a computation that uses it. Prior work has shown that QIF techniques based on *abstract interpretation* with *probabilistic polyhedra* can be used to analyze the worst-case leakage of a query, on-line, to determine whether that query can be safely answered. While this approach can provide precise estimates, it does not scale well. This paper shows how to solve the scalability problem by augmenting the baseline technique with *sampling* and *symbolic execution*. We prove that our approach never underestimates a query's leakage (it is sound), and detailed experimental results show that we can match the precision of the baseline technique but with orders of magnitude better performance.

1 Introduction

As more sensitive data is created, collected, and analyzed, we face the problem of how to productively use this data while preserving privacy. One approach to this problem is to analyze a query f in order to *quantify* how much information about secret input s is leaked by the output $f(s)$. More precisely, we can consider a querier to have some *prior belief* of the secret's possible values. The belief can be modeled as a probability distribution [10], i.e., a function δ from each possible value of s to its probability. When a querier observes output $o = f(s)$, he *revises* his belief, using Bayesian inference, to produce a *posterior* distribution δ'. If the posterior could reveal too much about the secret, then the query should be rejected. One common definition of "too much" is *Bayes Vulnerability*, which is the probability of the adversary guessing the secret in one try [41]. Formally,

$$V(\delta) \stackrel{\text{def}}{=} \max_i \delta(i)$$

Various works [6, 19, 24, 25] propose rejecting f if there exists an output that makes the vulnerability of the posterior exceed a fixed threshold K. In particular, for all possible values i of s (i.e., $\delta(i) > 0$), if the output $o = f(i)$ could induce a posterior δ' with $V(\delta') > K$, then the query is rejected.

One way to implement this approach is to estimate $f(\delta)$—the distribution of f's outputs when the inputs are distributed according to δ—by viewing f as a program in a *probabilistic programming language* (PPL) [18]. Unfortunately,

as discussed in Section 9, most PPLs are approximate in a manner that could easily result in *underestimating* the vulnerability, leading to an unsafe security decision. Techniques designed specifically to quantify information leakage often assume only uniform priors, cannot compute vulnerability (favoring, for example, Shannon entropy), and/or cannot maintain assumed knowledge between queries.

Mardziel et al. [25] propose a *sound* analysis technique based on abstract interpretation [12]. In particular, they estimate a program's probability distribution using an abstract domain called a *probabilistic polyhedron* (PP), which pairs a standard numeric abstract domain, such as *convex polyhedra* [13], with some additional *ornaments*, which include lower and upper bounds on the size of the support of the distribution, and bounds on the probability of each possible secret value. Using PP can yield a precise, yet safe, estimate of the vulnerability, and allows the posterior PP (which is not necessarily uniform) to be used as a prior for the next query. Unfortunately, PPs can be very inefficient. Defining *intervals* [11] as the PP's numeric domain can dramatically improve performance, but only with an unacceptable loss of precision.

In this paper we present a new approach that ensures a better balance of both precision and performance in vulnerability computation, augmenting PP with two new techniques. In both cases we begin by analyzing a query using the fast interval-based analysis. Our first technique is then to use *sampling* to augment the result. In particular, we execute the query using possible secret values i sampled from the posterior δ' derived from a particular output o_i. If the analysis were perfectly accurate, executing $f(i)$ would produce o_i. But since intervals are overapproximate, sometimes it will not. With many sampled outcomes, we can construct a Beta distribution to estimate the size of the support of the posterior, up to some level of confidence. We can use this estimate to boost the lower bound of the abstraction, and thus improve the precision of the estimated vulnerability.

Our second technique is of a similar flavor, but uses symbolic reasoning to magnify the impact of a successful sample. In particular, we execute a query result-consistent sample *concolically* [39], thus maintaining a symbolic formula (called the *path condition*) that characterizes the set of variable valuations that would cause execution to follow the observed path. We then count the number of possible solutions and use the count to boost the lower bound of the support (with 100% confidence).

Sampling and concolic execution can be combined for even greater precision.

We have formalized and proved our techniques are sound (Sections 3–6) and implemented and evaluated them (Sections 7 and 8). Using a privacy-sensitive ship planning scenario (Section 2) we find that our techniques provide similar precision to convex polyhedra while providing orders-of-magnitude better performance. More experiments are needed to see if the approach provides such benefits more generally. Our implementation freely available at https://github.com/GaloisInc/TAMBA.

Field	Type	Range	Private?
ShipID	Integer	1–10	No
NationID	Integer	1–20	No
Capacity	Integer	0–1000	Yes
Latitude	Integer	-900,000–900,000	Yes
Longitude	Integer	-1,800,000–1,800,000	Yes

Fig. 1. The data model used in the evacuation scenario.

2 Overview

To provide an overview of our approach, we will describe the application of our techniques to a scenario that involves a coalition of ships from various nations operating in a shared region. Suppose a natural disaster has impacted some islands in the region. Some number of individuals need to be evacuated from the islands, and it falls to a regional disaster response coordinator to determine how to accomplish this. While the coalition wants to collaborate to achieve these humanitarian aims, we assume that each nation also wants to protect their sensitive data—namely ship locations and capacity.

More formally, we assume the use of the data model shown in Figure 1, which considers a set of ships, their coalition affiliation, the evacuation capacity of the ship, and its position, given in terms of latitude and longitude.[1] We sometimes refer to the latter two as a location L, with $L.x$ as the longitude and $L.y$ as the latitude. We will often index properties by ship ID, writing Capacity(z) for the capacity associated with ship ID z, or Location(z) for the location.

The **evacuation problem** is defined as follows

> Given a target location L and number of people to evacuate N, compute a set of nearby ships S such that $\sum_{z \in S} \text{Capacity}(z) \geq N$.

Our goal is to solve this problem in a way that minimizes the vulnerability to the coordinator of private information, i.e., the ship locations and their exact capacity. We assume that this coordinator initially has no knowledge of the positions or capabilities of the ships other than that they fall within certain expected ranges.

If all members of the coalition share all of their data with the coordinator, then a solution is easy to compute, but it affords no privacy. Figure 2 gives an algorithm the response coordinator can follow that does not require each member to share all of their data. Instead, it iteratively performs queries *AtLeast* and *Nearby*. These queries do not reveal precise values about ship locations or capacity, but rather admit ranges of possibilities. The algorithm works by maintaining upper and lower bounds on the capacity of each ship `i` in the array `berths`. Each ship's bounds are updated based on the results of queries about its capacity and location. These queries aim to be privacy preserving, doing a sort of binary search to narrow in on the capacity of each ship in the operating area. The procedure completes once `is_solution` determines the minimum required capacity is reached.

[1] We give latitude and longitude values as integer representations of *decimal degrees* fixed to four decimal places; e.g., 14.3579 decimal degrees is encoded as 143579.

```
(* S = #ships; N = #evacuees; L = island loc.; D = min. proximity to L *)
    let berths = Array.make S (0,1000)
    let is_solution () = sum (Array.map fst berths) ≥ N
    let mid (x,y) = (x + y) / 2
    let AtLeast(z,b) = Capacity(z) ≥ b
    let Nearby(z,l,d) = |Loc(z).x − l.x| + |Loc(z).y − l.y| ≤ d
    while true do
      for i = 0 to S do
        let ask = mid berths[i]
        let ok = AtLeast(i,ask) && Nearby(i,L,D)
        if ok then  berths[i] ← (ask, snd berths[i])
        else        berths[i] ← (fst berths[i], ask)
        if is_solution () then return berths
      done
    done
```

Fig. 2. Algorithm to solve the evacuation problem for a single island.

2.1 Computing vulnerability with abstract interpretation

Using this procedure, what is revealed about the private variables (location and capacity)? Consider a single $\mathit{Nearby}(z, l, d)$ query. At the start, the coordinator is assumed to know only that z is somewhere within the operating region. If the query returns true, the coordinator now knows that s is within d units of l (using Manhattan distance). This makes $\mathit{Location}(z)$ more vulnerable because the adversary has less uncertainty about it.

Mardziel et al. [25] proposed a static analysis for analyzing queries such as $\mathit{Nearby}(z, l, d)$ to estimate the worst-case vulnerability of private data. If the worst-case vulnerability is too great, the query can be rejected. A key element of their approach is to perform abstract interpretation over the query using an abstract domain called a *probabilistic polyhedron*. An element P of this domain represents the set of possible distributions over the query's state. This state includes both the hidden secrets and the visible query results. The abstract interpretation is sound in the sense that the true distribution δ is contained in the set of distributions represented by the computed probabilistic polyhedron P.

A probabilistic polyhedron P is a tuple comprising a *shape* and three *ornaments*. The shape C is an element of a standard numeric domain—e.g., intervals [11], octagons [29], or convex polyhedra [13]—which overapproximates the set of possible values in the support of the distribution. The ornaments $p \in [0, 1]$, $m \in \mathbb{R}$, and $s \in \mathbb{Z}$ are pairs which store upper and lower bounds on the probability per point, the total mass, and number of support points in the distribution, respectively. (Distributions represented by P are not necessarily normalized, so the mass m is not always 1.)

Figure 3(a) gives an example probabilistic polyhedron that represents the posterior of a *Nearby* query that returns true. In particular, if *Nearby(z,*L_1*,D)* is true then Location(z) is somewhere within the depicted diamond around

L_1. Using convex polyhedra or octagons for the shape domain would permit representing this diamond exactly; using intervals would overapproximate it as the depicted 9x9 bounding box. The ornaments would be the same in any case: the size s of the support is 41 possible (x,y) points, the probability p per point is 0.01, and the total mass is 0.41, i.e., $p \cdot s$. In general, each ornament is a pair of a lower and upper bound (e.g., $s_{\min}$ and $s_{\max}$), and m might be a more accurate estimate than $p \cdot s$. In this case shown in the figure, the bounds are tight.

Mardziel et al's procedure works by computing the posterior P for each possible query output o, and from that posterior determining the vulnerability. This is easy to do. The upper bound $p_{\max}$ of p maximizes the probability of any given point. Dividing this by the *lower bound* $m_{\min}$ of the probability mass m normalizes this probability for the worst case. For P shown in Figure 3(a), the bounds of p and m are tight, so the vulnerability is simply $0.01/0.41 = 0.024$.

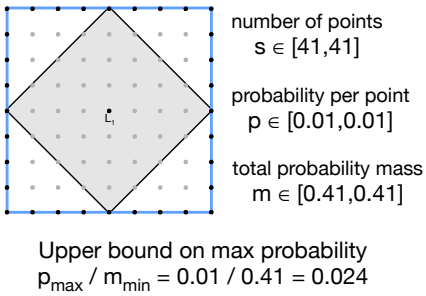
2.2 Improving precision with sampling and concolic execution

In Figure 3(a), the parameters s, p, and m are precise. However, as additional operations are performed, these quantities can accumulate imprecision. For example, suppose we are using intervals for the shape domain, and we wish to analyze the query $\mathit{Nearby}(z, L_1, 4) \vee \mathit{Nearby}(z, L_2, 4)$ (for some nearby point L_2). The result is produced by analyzing the two queries separately and then combining them with an *abstract join*; this is shown in the top row of Figure 3(b). Unfortunately, the result is very imprecise. The bottom row of Figure 3(b) illustrates the result we would get by using convex polyhedra as our shape domain. When using intervals (top row), the vulnerability is estimated as 0.036, whereas the precise answer (bottom row) is actually 0.026. Unfortunately, obtaining this precise answer is far more expensive than obtaining the imprecise one.

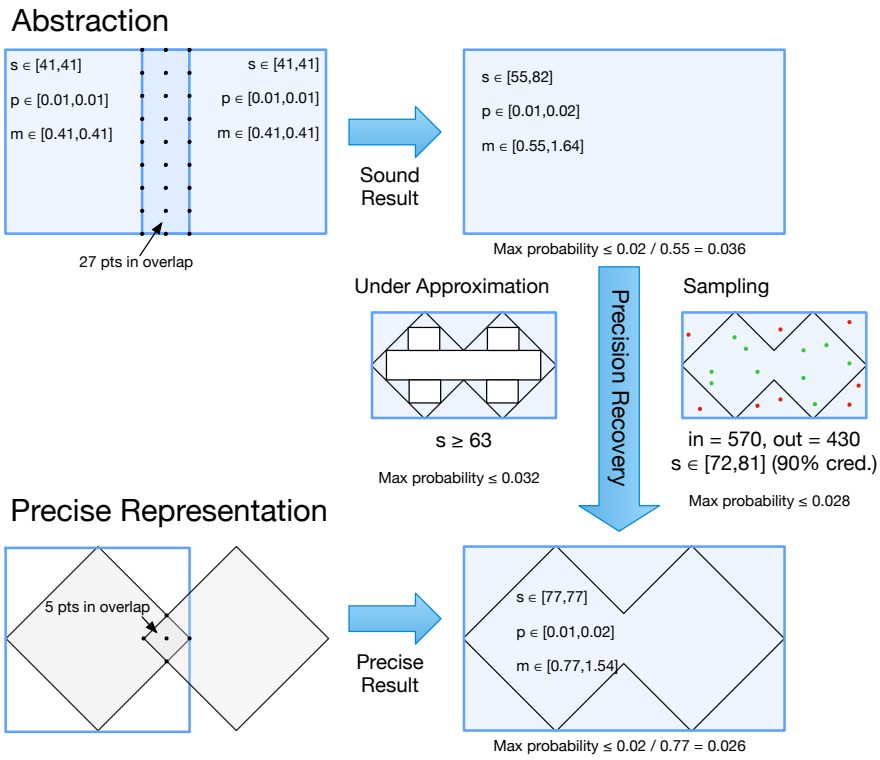
This paper presents two techniques that can allow us to use the less precise interval domain but then *recover* lost precision in a relatively cheap post-processing step. The effect of our techniques is shown in the middle-right of Figure 3(b). Both techniques aim to obtain better lower bounds for s. This allows us to update lower bounds on the probability mass m since $m_{\min}$ is at least $s_{\min} \cdot p_{\min}$ (each point has at least probability $p_{\min}$ and there are at least $s_{\min}$ of them). A larger m means a smaller vulnerability.

The first technique we explore is *sampling*, depicted to the right of the arrow in Figure 3(b). Sampling chooses random points and evaluates the query on them to determine whether they are in the support of the posterior distribution for a particular query result. By tracking the ratio of points that produce the expected output, we can produce an estimate of s, whose confidence increases as we include more samples. This approach is depicted in the figure, where we conclude that $s \in [72, 81]$ and $m \in [0.72, 1.62]$ with 90% confidence after taking 1000 samples, improving our vulnerability estimate to $V \leq \frac{0.02}{0.72} = 0.028$.

The second technique we explore is the use of *concolic execution* to derive a *path condition*, which is a formula over secret values that is consistent with a query result. By performing *model counting* to estimate the number of solutions to this formula, which are an underapproximation of the true size of the distribution, we



(a) Probabilistic polyhedra



(b) Improving precision with sampling and underapproximation (concolic execution)

Fig. 3. Computing vulnerability (max probability) using abstract interpretation

$$
\begin{array}{llll}
\textit{Variables} & x & \in & \mathbf{Var} \\
\textit{Integers} & n & \in & \mathbb{Z} \\
\textit{Rationals} & q & \in & \mathbb{Q} \\
\textit{States} & \sigma & \in & \mathbf{State} \stackrel{\text{def}}{=} \mathbf{Var} \rightharpoonup \mathbb{Z} \\
\textit{Distributions} & \delta & \in & \mathbf{Dist} \stackrel{\text{def}}{=} \mathbf{State} \to \mathbb{R}{+}_0 \\
\textit{Arith.ops} & \mathit{aop} & ::= & + \mid \times \mid - \\
\textit{Rel.ops} & \mathit{relop} & ::= & \leq \mid < \mid = \mid \neq \mid \cdots \\
\textit{Arith.exps} & E & ::= & x \mid n \mid E_1 \ \mathit{aop} \ E_2 \\
\textit{Bool.exps} & B & ::= & E_1 \ \mathit{relop} \ E_2 \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B \\
\textit{Statements} & S & ::= & \mathsf{skip} \mid x \ := \ E \mid S_1 \ ; \ S_2 \mid \mathsf{while} \ B \ \mathsf{do} \ S \mid \\
 & & & \quad \mathsf{if} \ B \ \mathsf{then} \ S_1 \ \mathsf{else} \ S_2 \mid \mathsf{pif} \ q \ \mathsf{then} \ S_1 \ \mathsf{else} \ S_2
\end{array}
$$

Fig. 4. Core language syntax

can safely boost the lower bound of s. This approach is depicted to the left of the arrow in Figure 3(b). The depicted shapes represent discovered path condition's disjuncts, whose size sums to 63. This is a better lower bound on s and improves the vulnerability estimate to 0.032.

These techniques can be used together to further increase precision. In particular, we can first perform concolic execution, and then sample from the area not covered by this underapproximation. Importantly, Section 8 shows that using our techniques with the interval-based analysis yields an orders of magnitude performance improvement over using polyhedra-based analysis alone, while achieving similar levels of precision, with high confidence.

3 Preliminaries: Syntax and Semantics

This section presents the core language—syntax and semantics—in which we formalize our approach to computing vulnerability. We also review *probabilistic polyhedra* [25], which is the baseline analysis technique that we augment.

3.1 Core Language and Semantics

The programming language we use for queries is given in Figure 4. The language is essentially standard, apart from $\mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$, which implements probabilistic choice: S_1 is executed with probability q, and S_2 with probability $1 - q$. We limit the form of expressions E so that they can be approximated by standard numeric abstract domains such as convex polyhedra [13]. Such domains require linear forms; e.g., there is no division operator and multiplication of two variables is disallowed.[2]

We define the semantics of a program in terms of its effect on (discrete) distributions of states. States σ are partial maps from variables to integers; we

[2] Relaxing such limitations is possible—e.g., polynominal inequalities can be approximated using convex polyhedra [5]—but doing so precisely and scalably is a challenge.

write $\mathit{domain}(\sigma)$ for the set of variables over which σ is defined. Distributions δ are maps from states to nonnegative real numbers, interpreted as probabilities (in range $[0, 1]$). The denotational semantics considers a program as a relation between distributions. In particular, the semantics of statement S, written $[\![S]\!]$, is a function of the form $\mathbf{Dist} \to \mathbf{Dist}$; we write $[\![S]\!]\delta = \delta'$ to say that the semantics of S maps input distribution δ to output distribution δ'. Distributions are not necessarily normalized; we write $\|\delta\|$ as the probability mass of δ (which is between 0 and 1). We write $\dot{\sigma}$ to denote the point distribution that gives σ probability 1, and all other states 0.

The semantics is standard and not crucial in order to understand our techniques. In Appendix B we provide the semantics in full. See Clarkson et al. [10] or Mardziel et al [25] for detailed explanations.

3.2 Probabilistic polyhedra

To compute vulnerability for a program S we must compute (an approximation of) its output distribution. One way to do that would be to use sampling: Choose states σ at random from the input distribution δ, "run" the program using that input state, and collect the frequencies of output states σ' into a distribution δ'. While using sampling in this manner is simple and appealing, it could be both expensive and imprecise. In particular, depending on the size of the input and output space, it may take many samples to arrive at a proper approximation of the output distribution.

Probabilistic polyhedra [25] can address both problems. This abstract domain combines a standard domain C for representing numeric program states with additional *ornaments* that all together can safely represent S's output distribution.

Probabilistic polyhedra work for any numeric domain; in this paper we use both convex polyhedra [13] and intervals [11]. For concreteness, we present the defintion using convex polyhedra. We use the meta-variables β, β_1, β_2, etc. to denote linear inequalities.

Definition 1. *A* convex polyhedron $C = (B, V)$ *is a set of linear inequalities* $B = \{\beta_1, \ldots, \beta_m\}$*, interpreted conjunctively, over variables* V*. We write* $\mathbb{C}$ *for the set of all convex polyhedra. A polyhedron* C *represents a set of states, denoted* $\gamma_{\mathbb{C}}(C)$*, as follows, where* $\sigma \models \beta$ *indicates that the state* σ *satisfies the inequality* β*.*

$$\gamma_{\mathbb{C}}((B, V)) \stackrel{\text{def}}{=} \{\sigma \ : \ \mathit{domain}(\sigma) = V,\ \forall \beta \in B.\ \sigma \models \beta\}$$

Naturally we require that $\mathit{domain}(\{\beta_1, \ldots, \beta_n\}) \subseteq V$*; i.e.,* V *mentions all variables in the inequalities. Let* $\mathit{domain}((B, V)) = V$*.*

Probabilistic polyhedra extend this standard representation of sets of program states to sets of *distributions* over program states.

Definition 2. *A* probabilistic polyhedron P *is a tuple* $(C, \mathrm{s}^{\min}, \mathrm{s}^{\max}, \mathrm{p}^{\min}, \mathrm{p}^{\max}, \mathrm{m}^{\min}, \mathrm{m}^{\max})$*. We write* $\mathbb{P}$ *for the set of probabilistic polyhedra. The quantities* $\mathrm{s}^{\min}$ *and* $\mathrm{s}^{\max}$ *are lower and upper bounds on the number of support points in*

the concrete distribution(s) P represents. A support point of a distribution is one which has non-zero probability. The quantities $\mathrm{p}^{\min}$ *and* $\mathrm{p}^{\max}$ *are lower and upper bounds on the probability mass per support point. The* $\mathrm{m}^{\min}$ *and* $\mathrm{m}^{\max}$ *components give bounds on the total probability mass (i.e., the sum of the probabilities of all support points). Thus P represents the* set *of distributions $\gamma_{\mathbb{P}}(P)$ defined below.*

$$\begin{aligned}\gamma_{\mathbb{P}}(P) \stackrel{\text{def}}{=} \{\delta \;:\; & \mathit{support}(\delta) \subseteq \gamma_{\mathbb{C}}(C) \wedge \\ & \mathrm{s}^{\min} \leq |\mathit{support}(\delta)| \leq \mathrm{s}^{\max} \wedge \\ & \mathrm{m}^{\min} \leq \|\delta\| \leq \mathrm{m}^{\max} \wedge \\ & \forall \sigma \in \mathit{support}(\delta).\ \mathrm{p}^{\min} \leq \delta(\sigma) \leq \mathrm{p}^{\max}\}\end{aligned}$$

We will write $\mathit{domain}(P) \stackrel{\text{def}}{=} \mathit{domain}(C)$ to denote the set of variables used in the probabilistic polyhedron.

Note the set $\gamma_{\mathbb{P}}(P)$ is a singleton exactly when $\mathrm{s}^{\min} = \mathrm{s}^{\max} = \#(C)$ and $\mathrm{p}^{\min} = \mathrm{p}^{\max}$, and $\mathrm{m}^{\min} = \mathrm{m}^{\max}$, where $\#(C)$ denotes the number of discrete points in convex polyhedron C. In such a case $\gamma_{\mathbb{P}}(P)$ contains only the uniform distribution where each state in $\gamma_{\mathbb{C}}(C)$ has probability $\mathrm{p}^{\min}$. In general, however, the concretization of a probabilistic polyhedron will have an infinite number of distributions, with per-point probabilities varied somewhere in the range $\mathrm{p}^{\min}$ and $\mathrm{p}^{\max}$. Distributions represented by a probabilistic polyhedron are not necessarily normalized. In general, there is a relationship between $\mathrm{p}^{\min}, \mathrm{s}^{\min}$, and $\mathrm{m}^{\min}$, in that $\mathrm{m}^{\min} \geq \mathrm{p}^{\min} \cdot \mathrm{s}^{\min}$ (and $\mathrm{m}^{\max} \leq \mathrm{p}^{\max} \cdot \mathrm{s}^{\max}$), and the combination of the three can yield more information than any two in isolation.

The *abstract semantics* of S is written $\langle\!\langle S \rangle\!\rangle\, P = P'$, and indicates that abstractly interpreting S where the distribution of input states are approximated by P will produce P', which approximates the distribution of output states. Following standard abstract interpretation terminology, $\wp\mathbf{Dist}$ (sets of distributions) is the *concrete domain*, $\mathbb{P}$ is the *abstract domain*, and $\gamma_{\mathbb{P}} : \mathbb{P} \to \wp\mathbf{Dist}$ is the *concretization function* for $\mathbb{P}$. We do not present the abstract semantics here; details can be found in Mardziel et al. [25]. Importantly, this abstract semantics is sound:

Theorem 1 (Soundness). *For all $S, P_1, P_2, \delta_1, \delta_2$, if $\delta_1 \in \gamma_{\mathbb{P}}(P_1)$ and $\langle\!\langle S \rangle\!\rangle\, P_1 = P_2$, then $[\![S]\!]\delta_1 = \delta_2$ with $\delta_2 \in \gamma_{\mathbb{P}}(P_2)$.*

Proof. See Theorem 6 in Mardziel et. al [25].

Consider the example from Section 2.2. We assume the adversary has no prior information about the location of ship s. So, δ_1 above is simply the uniform distribution over all possible locations. The statement S is the query issued by the adversary, $\mathit{Nearby}(z, L_1, 4) \vee \mathit{Nearby}(z, L_2, 4)$.[3] If we assume that the result of the query is `true` then the adversary learns that the location of s is within (Manhattan) distance 4 of L_1 or L_2. This posterior belief (δ_2) is represented

[3] Appendix A shows the code, which computes Manhattan distance between s and L_1 and L_2 and then sets an output variable if either distance is within four units.

by the overlapping diamonds on the bottom-right of Figure 3(b). The abstract interpretation produces a sound (interval) overapproximation (P_2) of the posterior belief. This is modeled by the rectangle which surrounds the overlapping diamonds. This rectangle is the "join" of two overlapping boxes, which each correspond to one of the *Nearby* calls in the disjuncts of S.

4 Computing Vulnerability: Basic procedure

The key goal of this paper is to quantify the risk to secret information of running a query over that information. This section explains the basic approach by which we can use probabilistic polyhedra to compute *vulnerability*, i.e., the probability of the most probable point of the posterior distribution. Improvements on this basic approach are given in the next two sections.

Our convention will be to use C_1, $\mathrm{s}_1^{\min}$, $\mathrm{s}_1^{\max}$, etc. for the components associated with probabilistic polyhedron P_1. In the program S of interest, we assume that secret variables are in the set T, so input states are written σ_T, and we assume there is a single output variable r. We assume that the adversary's initial uncertainty about the possible values of the secrets T is captured by the probabilistic polyhedron P_0 (such that $\mathit{domain}(P_0) \supseteq T$).

Computing vulnerability occurs according to the following procedure.

1. Perform abstract interpretation: $\langle\!\langle S \rangle\!\rangle\, P_0 = P$
2. Given a concrete output value of interest, o, perform abstract conditioning to define $P_{r=o} \stackrel{\text{def}}{=} (P \wedge r = o)$.[4]

The vulnerability V is the probability of the most likely state(s). When a probabilistic polyhedron represents one or more true distributions (i.e., the probabilities all sum to 1), the most probable state's probability is bounded by $\mathrm{p}^{\max}$. However, the abstract semantics does not always normalize the probabilistic polyhedron as it computes, so we need to scale $\mathrm{p}^{\max}$ according to the total probability mass. To ensure that our estimate is on the safe side, we scale $\mathrm{p}^{\max}$ using the *minimum* probability mass: $V = \frac{\mathrm{p}^{\max}}{\mathrm{m}^{\min}}$. In Figure 3(b), the sound approximation in the top-right has $V \leq \frac{0.02}{0.55} = 0.036$ and the most precise approximation in the bottom-right has $V \leq \frac{0.02}{0.77} = 0.026$.

5 Improving precision with sampling

We can improve the precision of the basic procedure using sampling. First we introduce some notational convenience:

$$P_T \stackrel{\text{def}}{=} P \wedge (r = o) \downharpoonright T$$
$$P_{T+} \stackrel{\text{def}}{=} P_T \text{ revised polyhedron with confidence } \omega$$

[4] We write $P \wedge B$ and not $P \mid B$ because P need not be normalized.

P_T is equivalent to step 2, above, but projected onto the set of secret variables T. P_{T+} is the improved (via sampling) polyhedron.

After computing P_T with the basic procedure from the previous section we take the following additional steps:

1. Set counters α and β to zero.
2. Do the following N times (for some N, see below):
   (a) Randomly select an input state $\sigma_T \in \gamma_{\mathbb{C}}(C_T)$.
   (b) "Run" the program by computing $[\![S]\!]\dot{\sigma_T} = \delta$. If there exists $\sigma \in \mathit{support}(\delta)$ with $\sigma(r) = o$ then increment α, else increment β.
3. We can interpret α and β as the parameters of a Beta distribution of the likelihood that an arbitrary state in $\gamma_{\mathbb{C}}(C_T)$ is in the support of the true distribution. From these parameters we can compute the *credible interval* $[p_L, p_U]$ within which is contained the true likelihood, with confidence ω (where $0 \leq \omega \leq 1$). A credible interval is essentially a Bayesian analogue of a confidence interval and can be computed from the cumulative distribution function (CDF) of the Beta distribution (the 99% credible interval is the interval $[a, b]$ such that the CDF at a has value 0.005 and the CDF at b has value 0.995). In general, obtaining a higher confidence or a narrower interval will require a higher N. Let result $P_{T+} = P_T$ except that $\mathrm{s}^{\min}_{T+} = p_L \cdot \#(C_T)$ and $\mathrm{s}^{\max}_{T+} = p_U \cdot \#(C_T)$ (assuming these improve on $\mathrm{s}^{\min}_T$ and $\mathrm{s}^{\max}_T$). We can then propagate these improvements to $\mathrm{m}^{\min}$ and $\mathrm{m}^{\max}$ by defining $\mathrm{m}^{\min}_{T+} = \mathrm{p}^{\min}_T \cdot \mathrm{s}^{\min}_{T+}$ and $\mathrm{m}^{\max}_{T+} = \mathrm{p}^{\max}_T \cdot \mathrm{s}^{\max}_{T+}$. Note that if $\mathrm{m}^{\min}_T > \mathrm{m}^{\min}_{T+}$ we leave it unchanged, and do likewise if $\mathrm{m}^{\max}_T < \mathrm{m}^{\max}_{T+}$.

At this point we can compute the vulnerability as in the basic procedure, but using P_{T+} instead of P_T.

Consider the example of Section 2.2. In Figure 3(b), we draw samples from the rectangle in the top-right. This rectangle overapproximates the set of locations where s might be, given that the query returned true. We sample locations from this rectangle and run the query on each sample. The green (red) dots indicate true (false) results, which are added to α (β). After sampling $N = 1000$ locations, we have $\alpha = 570$ and $\beta = 430$. Choosing $\omega = .9$ (90%), we compute the credible interval $[0.53, 0.60]$. With $\#(C_T) = 135$, we compute $[\mathrm{s}^{\min}_{T+}, \mathrm{s}^{\max}_{T+}]$ as $[0.53 \cdot 135, 0.60 \cdot 135] = [72, 81]$.

There are several things to notice about this procedure. First, observe that in step 2b we "run" the program using the point distribution $\dot{\sigma}$ as an input; in the case that S is deterministic (has no pif statements) the output distribution will also be a point distribution. However, for programs with pif statements there are multiple possible outputs depending on which branch is taken by a pif. We consider all of these outputs so that we can confidently determine whether the input state σ could ever cause S to produce result o. If so, then σ should be considered part of P_{T+}. If not, then we can safely rule it out (i.e., it is part of the overapproximation).

Second, we only update the size parameters of P_{T+}; we make no changes to $\mathrm{p}^{\min}_{T+}$ and $\mathrm{p}^{\max}_{T+}$. This is because our sampling procedure only determines whether it is *possible* for an input state to produce the expected output. The probability

that an input state produces an output state is already captured (soundly) by p_T so we do not change that. This is useful because the approximation of p_T does not degrade with the use of the interval domain in the way the approximation of the size degrades (as illustrated in Figure 3(b)). Using sampling is an attempt to regain the precision lost on the size component (only).

Finally, the confidence we have that sampling has accurately assessed which input states are in the support is orthogonal to the probability of any given state. In particular, P_T is an abstraction of a distribution δ_T, which is a mathematical object. Confidence ω is a measure of how likely it is that our abstraction (or, at least, the size part of it) is accurate.

We prove (in our extended report [43]) that our sampling procedure is sound:

Theorem 2 (Sampling is Sound).
If $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$, $\langle\!\langle S \rangle\!\rangle\, P_0 = P$, *and* $[\![S]\!]\delta_0 = \delta$ *then* $\delta_T \in \gamma_{\mathbb{P}}(P_{T+})$ *with confidence* ω *where*

$$\begin{aligned} \delta_T &\stackrel{\text{def}}{=} \delta \wedge (r = o) \lfloor T \\ P_T &\stackrel{\text{def}}{=} P \wedge (r = o) \lfloor T \\ P_{T+} &\stackrel{\text{def}}{=} P_T \text{ sampling revised with confidence } \omega. \end{aligned}$$

6 Improving precision with concolic execution

Another approach to improving the precision of a probabilistic polyhedron P is to use concolic execution. The idea here is to "magnify" the impact of a single sample to soundly increase $\mathrm{s}^{\min}$ by considering its execution *symbolically*. More precisely, we concretely execute a program using a particular secret value, but maintain symbolic constraints about how that value is used. This is referred to as *concolic* execution [39]. We use the collected constraints to identify all points that would induce the same execution path, which we can include as part of $\mathrm{s}^{\min}$.

We begin by defining the semantics of concolic execution, and then show how it can be used to increase $\mathrm{s}^{\min}$ soundly.

6.1 (Probabilistic) Concolic Execution

Concolic execution is expressed as rewrite rules defining a judgment $\langle \Pi, S \rangle \longrightarrow^{p}_{\pi} \langle \Pi', S' \rangle$. Here, Π is pair consisting of a concrete state σ and symbolic state ζ. The latter maps variables $x \in \mathbf{Var}$ to *symbolic expressions* $\mathcal{E}$ which extend expressions E with *symbolic variables* α. This judgment indicates that under input state Π the statement S reduces to statement S' and output state Π' with probability p, with *path condition* π. The path condition is a conjunction of boolean symbolic expressions $\mathcal{B}$ (which are just boolean expressions B but altered to use symbolic expressions $\mathcal{E}$ instead of expressions E) that record which branch is taken during execution. For brevity, we omit π in a rule when it is true.

The rules for the concolic semantics are given in Figure 5. Most of these are standard, and deterministic (the probability annotation p is 1). Path conditions

$$\langle(\sigma,\zeta), x := E\rangle \longrightarrow^1 \langle(\sigma[x \mapsto \sigma(E)], \zeta[x \mapsto \zeta(E)]), \mathsf{skip}\rangle$$
$$\langle(\sigma,\zeta), \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^1_{\zeta(B)} \langle(\sigma,\zeta), S_1\rangle \quad \text{if } \sigma(B)$$
$$\langle(\sigma,\zeta), \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^1_{\zeta(\neg B)} \langle(\sigma,\zeta), S_2\rangle \text{ if } \sigma(\neg B)$$
$$\langle \Pi, \mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^q \langle \Pi, S_1\rangle$$
$$\langle \Pi, \mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \longrightarrow^{1-q} \langle \Pi, S_2\rangle$$
$$\langle \Pi, S_1\ ;\ S_2\rangle \longrightarrow^1_\pi \langle \Pi', S_1'\ ;\ S_2\rangle \quad \text{if } \langle \Pi, S_1\rangle \longrightarrow^1_\pi \langle \Pi', S_1'\rangle$$
$$\langle \Pi, \mathsf{skip}\ ;\ S\rangle \longrightarrow^1 \langle \Pi, S\rangle$$
$$\langle \Pi, \mathsf{while}\ B\ \mathsf{do}\ S\rangle \longrightarrow^1_{\zeta(B)} \langle \Pi, S\ ;\ \mathsf{while}\ B\ \mathsf{do}\ S\rangle \quad \text{if } \sigma(B)$$
$$\langle \Pi, \mathsf{while}\ B\ \mathsf{do}\ S\rangle \longrightarrow^1_{\zeta(\neg B)} \langle \Pi, \mathsf{skip}\rangle \quad \text{if } \sigma(\neg B)$$

Fig. 5. Concolic semantics

are recorded for if and while, depending on the branch taken. The semantics of pif q then S_1 else S_2 is non-deterministic: the result is that of S_1 with probability q, and S_2 with probability $1-q$. We write $\zeta(B)$ to substitute free variables $x \in B$ with their mapped-to values $\zeta(x)$ and then simplify the result as much as possible. For example, if $\zeta(x) = \alpha$ and $\zeta(y) = 2$, then $\zeta(x > y + 3) = \alpha > 5$. The same goes for $\zeta(E)$.

We define a *complete run* of the concolic semantics with the judgment $\langle \Pi, S\rangle \Downarrow^p_\pi \Pi'$, which has two rules:

$$\langle \Pi, \mathsf{skip}\rangle \Downarrow^1_{\mathsf{true}} \Pi$$

$$\frac{\langle \Pi, S\rangle \longrightarrow^p_\pi \langle \Pi', S'\rangle \quad \langle \Pi', S'\rangle \Downarrow^q_{\pi'} \Pi''}{\langle \Pi, S\rangle \Downarrow^{p\cdot q}_{\pi \wedge \pi'} \Pi''}$$

A complete run's probability is thus the product of the probability of each individual step taken. The run's path condition is the conjunction of the conditions of each step.

The path condition π for a complete run is a conjunction of the (symbolic) boolean guards evaluated during an execution. π can be converted to disjunctive normal form (DNF), and given the restrictions of the language the result is essentially a set of convex polyhedra over symbolic variables α.

6.2 Improving precision

Using concolic execution, we can improve our estimate of the size of a probabilistic polyhedron as follows:

1. Randomly select an input state $\sigma_T \in \gamma_{\mathbb{C}}(C_T)$ (recall that C_T is the polyhedron describing the possible valuations of secrets T).
2. Set $\Pi = (\sigma_T, \zeta_T)$ where ζ_T maps each variable $x \in T$ to a fresh symbolic variable α_x. Perform a complete concolic run $\langle \Pi, S\rangle \Downarrow^p_\pi (\sigma', \zeta')$. Make sure that $\sigma'(r) = o$, i.e., the expected output. If not, select a new σ_T and retry. Give up after some number of failures N. For our example shown in Figure 3(b), we might obtain a path condition $|Loc(z).x - L_1.x| + |Loc(z).y - L_1.y| \leq 4$ that captures the left diamond of the disjunctive query.

3. After a successful concolic run, convert path condition π to DNF, where each conjunctive clause is a polyhedron C_i. Also convert uses of disequality ($\leq$ and $\geq$) to be strict ($<$ and $>$).
4. Let $C = C_T \sqcap (\bigsqcup_i C_i)$; that is, it is the join of each of the polyhedra in $DNF(\pi)$ "intersected" with the original constraints. This captures all of the points that could possibly lead to the observed outcome along the concolically executed path. Compute $n = \#(C)$. Let $P_{T+} = P_T$ except define $\mathrm{s}^{\min}_{T+} = n$ if $\mathrm{s}^{\min}_{T} < n$ and $\mathrm{m}^{\min}_{T+} = \mathrm{p}^{\min}_{T} \cdot n$ if $\mathrm{m}^{\min}_{T} < \mathrm{p}^{\min}_{T} \cdot n$. (Leave them as is, otherwise.) For our example, $n = 41$, the size of the left diamond. We do not update $\mathrm{s}^{\min}_{T}$ since $41 < 55$, the probabilistic polyhedron's lower bound (but see below).

Theorem 3 (Concolic Execution is Sound).
If $\delta_0 \in \gamma_{\mathbb{P}}(P_0)$, $\langle\!\langle S \rangle\!\rangle\, P_0 = P$, and $[\![S]\!]\delta_0 = \delta$ then $\delta_T \in \gamma_{\mathbb{P}}(P_{T+})$ where

$$\begin{aligned} \delta_T &\stackrel{\text{def}}{=} \delta \wedge (r = o) \downharpoonright T \\ P_T &\stackrel{\text{def}}{=} P \wedge (r = o) \downharpoonright T \\ P_{T+} &\stackrel{\text{def}}{=} P_T \textit{ concolically revised.} \end{aligned}$$

The proof is in the extended technical report [43].

6.3 Combining Sampling with Concolic Execution

Sampling can be used to further augment the results of concolic execution. The key insight is that the presence of a sound under-approximation generated by the concolic execution means that it is unnecessary to sample from the under-approximating region. Here is the algorithm:

1. Let $C = C_0 \sqcap (\bigsqcup_i C_i)$ be the under-approximating region.
2. Perform sampling per the algorithm in Section 5, but with two changes:
   - if a sampled state $\sigma_T \in \gamma_{\mathbb{C}}(C)$, ignore it
   - When done sampling, compute $\mathrm{s}^{\min}_{T+} = p_L \cdot (\#(C_T) - \#(C)) + \#(C)$ and $\mathrm{s}^{\max}_{T+} = p_U \cdot (\#(C_T) - \#(C)) + \#(C)$. This differs from Section 5 in not including the count from concolic region C in the computation. This is because, since we ignored samples $\sigma_T \in \gamma_{\mathbb{C}}(C)$, the credible interval $[p_L, p_U]$ bounds the likelihood that any given point in $C_T \setminus C$ is in the support of the true distribution.

For our example, concolic execution indicated there are at least 41 points that satisfy the query. With this in hand, and using the same samples as shown in Section 5, we can refine $s \in [74, 80]$ and $m \in [0.74, 0.160]$ (the credible interval is formed over only those samples which satisfy the query but fall outside the under-approximation returned by concolic execution). We improve the vulnerability estimate to $V \leq \frac{0.02}{0.0.74} = 0.027$. These bounds (and vulnerability estimate) are better than those of sampling alone ($s \in [72, 81]$ with $V \leq 0.028$).

The statement of soundness and its proof can be found in the extended technical report [43].

7 Implementation

We have implemented our approach as an extension of Mardziel et al. [25], which is written in OCaml. This baseline implements numeric domains C via an OCaml interface to the Parma Polyhedra Library [4]. The counting procedure $\#(C)$ is implemented by LattE [15]. Support for arbitrary precision and exact arithmetic (e.g., for manipulating $\mathrm{m}^{\mathrm{min}}$, $\mathrm{p}^{\mathrm{min}}$, etc.) is provided by the `mlgmp` OCaml interface to the GNU Multi Precision Arithmetic library. Rather than maintaining a single probabilistic polyhedron P, the implementation maintains a *powerset* of polyhedra [3], i.e., a finite disjunction. Doing so results in a more precise handling of join points in the control flow, at a somewhat higher performance cost.

We have implemented our extensions to this baseline for the case that domain C is the interval numeric domain [11]. Of course, the theory fully applies to any numeric abstract domain. We use Gibbs sampling, which we implemented ourselves. We delegate the calculation of the beta distribution and its corresponding credible interval to the `cephes` OCaml library, which in turns uses the GNU Scientific Library. It is straightforward to lift the various operations we have described to the powerset domain. All of our code is available at https://github.com/GaloisInc/TAMBA.

8 Experiments

To evaluate the benefits of our techniques, we applied them to queries based on the evacuation problem outlined in Section 2. We found that while the baseline technique can yield precise answers when computing vulnerability, our new techniques can achieve close to the same level of precision far more efficiently.

8.1 Experimental Setup

For our experiments we analyzed queries similar to $\mathit{Nearby}(s, l, d)$ from Figure 2. We generalize the *Nearby* query to accept a set of locations L—the query returns true if s is within d units of any one of the islands having location $l \in L$. In our experiments we fix $d = 100$. We consider the secrecy of the location of s, $\mathit{Location}(s)$. We also analyze the execution of the resource allocation algorithm of Figure 2 directly; we discuss this in Section 8.3.

We measure the time it takes to compute the *vulnerability* (i.e., the probability of the most probable point) following each query. In our experiments, we consider a single ship s and set its coordinates so that it is always in range of some island in L, so that the concrete query result returns `true` (i.e. $\mathit{Nearby}(s, L, 100) = \mathit{true}$). We measure the vulnerability following this query result starting from a prior belief that the coordinates of s are uniformly distributed with $0 \leq \mathrm{Location}(s).x \leq 1000$ and $0 \leq \mathrm{Location}(s).y \leq 1000$.

In our experiments, we varied several experimental parameters: *analysis method* (either P, I, CE, S, or CE+S), *query complexity* c; *AI precision level* p; and *number of samples* n. We describe each in turn.

Analysis method We compared five techniques for computing vulnerability:
P: Abstract interpretation (AI) with convex polyhedra for domain C (Section 4),
I: AI with intervals for C (Section 4),
S: AI with intervals augmented with sampling (Section 5),
CE: AI with intervals augmented with concolic execution (Section 6), and
CE+S: AI with intervals augmented with both techniques (Section 6.3)

The first two techniques are due to Mardziel et al. [25], where the former uses convex polyhedra and the latter uses intervals (aka boxes) for the underlying polygons. In our experiments we tend to focus on P since I's precision is unacceptably poor (e.g., often vulnerability = 1).

Query complexity. We consider queries with different L; we say we are increasing the *complexity* of the query as L gets larger. Let $c = |L|$; we consider $1 \leq c \leq 5$, where larger L include the same locations as smaller ones. We set each location to be at least $2 \cdot d$ Manhattan distance units away from any other island (so diamonds like those in Figure 3(a) never overlap).

Precision. The precision parameter p bounds the size of the powerset abstract domain at all points during abstract interpretation. This has the effect of forcing joins when the powerset grows larger than the specified precision. As p grows larger, the results of abstract interpretation are likely to become more precise (i.e. vulnerability gets closer to the true value). We considered p values of 1, 2, 4, 8, 16, 32, and 64.

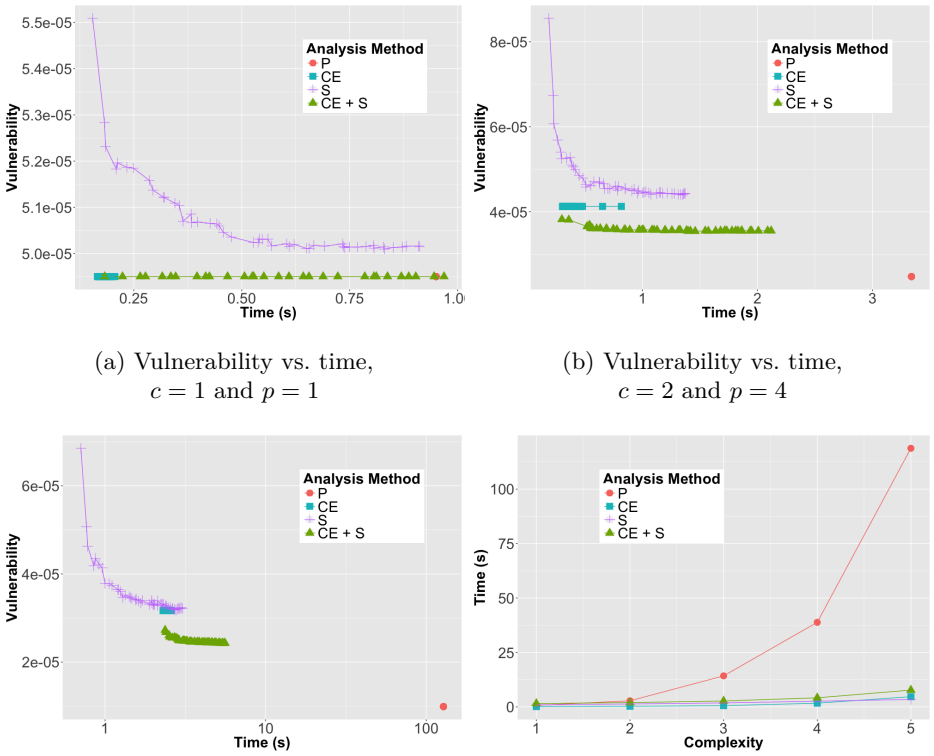
Samples taken. For the latter three analysis methods, we varied the number of samples taken n. For analysis CE, n is interpreted as the number of samples to try per polyhedron before giving up trying to find a "valid sample."[5] For analysis S, n is the number of samples, distributed proportionally across all the polyhedra in the powerset. For analysis CE+S, n is the combination of the two. We considered sample size values of $1,000 - 50,000$ in increments of $1,000$. We always compute an interval with $\omega = 99.9\%$ confidence (which will be wider when fewer samples are used).

System description. We ran experiments varying all possible parameters. For each run, we measured the total execution time (wall clock) in seconds to analyze the query and compute vulnerability. All experiments were carried out on a MacBook Air with OSX version 10.11.6, a 1.7GHz Intel Core i7, and 8GB of RAM. We ran a single trial for each configuration of parameters. Only wall-clock time varies across trials; informally, we observed time variations to be small.

8.2 Results

Figure 6(a)–(c) measure vulnerability (y-axis) as a function of time (x-axis) for each analysis.[6] These three figures characterize three interesting "zones" in the space of complexity and precision. The results for method I are not shown in any of the figures. This is because I always produces a vulnerability of 1. The refinement methods (CE, S, and CE+S) are all over the interval domain, and should be considered as "improving" the vulnerability of I.

[5] This is the N parameter from section 6.
[6] These are best viewed on a color display.



(a) Vulnerability vs. time, $c = 1$ and $p = 1$

(b) Vulnerability vs. time, $c = 2$ and $p = 4$

(c) Vulnerability vs. time, $c = 5$ and $p = 32$ (X-axis is log-scale)

(d) Time vs. complexity, $n = 50,000$ and $p = 64$

Fig. 6. Experimental results

In Figure 6(a) we fix $c = 1$ and $p = 1$. In this configuration, baseline analysis P can compute the true vulnerability in ~ 0.95 seconds. Analysis CE is also able to compute the true vulnerability, but in $\sim$0.19 seconds. Analysis S is able to compute a vulnerability to within $\sim 5 \cdot e^{-6}$ of optimal in ~ 0.15 seconds. These data points support two key observations. First, even a very modest number of samples improves vulnerability significantly over just analyzing with intervals. Second, concolic execution is only slightly slower and can achieve the optimal vulnerability. Of course, concolic execution is not a panacea. As we will see, a feature of this configuration is that no joins take place during abstract interpretation. This is critical to the precision of the concolic execution.

In Figure 6(b) we fix $c = 2$ and $p = 4$. In contrast to the configuration of Figure 6(a), the values for c and p in this configuration are not sufficient to prevent all joins during abstract interpretaion. This has the effect of taking polygons that represent individual paths through the program and joining them into a single polygon representing many paths. We can see that this is the case because baseline analysis P is now achieving a better vulnerability than CE. However, one

Table 1. Analyzing a 3-ship resource allocation run

Resource Allocation (3 ships)		
Analysis	Time (s)	Vulnerability
P	Timeout (5 min)	N/A
I	0.516	1
CE	16.650	$1.997 \cdot 10^{-24}$
S	1.487	$1.962 \cdot 10^{-24}$
CE+S	17.452	$1.037 \cdot 10^{-24}$

pattern from the previous configuration persists: all three refinement methods (CE, S, CE+S) can achieve vulnerability within $\sim 1 \cdot e^{-5}$ of P, but in $\frac{1}{4}$ the time. In contrast to the previous configuration, analysis CE+S is now able to make a modest improvement over CE (since it does not achieve the optimal).

In Figure 6(c) we fix $c = 5$ and $p = 32$. This configuration magnifies the effects we saw in Figure 6(b). Similarly, in this configuration there are joins happening, but the query is much more complex and the analysis is much more precise. In this figure, we label the X axis as a log scale over time. This is because analysis P took over two minutes to complete, in contrast the longest-running refinement method, which took less than 6 seconds. The relationship between the refinement analyses is similar to the previous configuration. The key observation here is that, again, all three refinement analyses achieve within $\sim 3 \cdot e^{-5}$ of P, but this time in 4% of the time (as opposed to $\frac{1}{4}$ in the previous configuration).

Figure 6(d) makes more explicit the relationship between refinements (CE, S, CE+S) and P. We fix $n = 50,000$ (the maximum) here, and $p = 64$ (the maximum). We can see that as query complexity goes up, P gets exponentially slower, while CE, S, and CE+S slow at a much lower rate, while retaining (per the previous graphs) similar precision.

8.3 Evacuation Problem

We conclude this section by briefly discussing an analysis of an execution of the resource allocation algorithm of Figure 2. In our experiment, we set the number of ships to be three, where two were in range $d = 300$ of the evacuation site, and their sum-total berths (500) were sufficient to satisfy demand at the site (also 500). For our analysis refinements we set $n = 1000$. Running the algorithm, a total of seven pairs of *Nearby* and *Capacity* queries were issued. In the end, the algorithm selects two ships to handle the evacuation.

Table 1 shows the time to execute the algorithm using the different analysis methods, along with the computed vulnerability—this latter number represents the coordinator's view of the most likely nine-tuple of the private data of the three ships involved (x coordinate, y coordinate, and capacity for each). We can see that, as expected, our refinement analyses are far more efficient than baseline P, and far more precise than baseline I. The CE methods are precise but slower

than S. This is because of the need to count the number of points in the DNF of the concolic path conditions, which is expensive.

9 Related Work

Quantifying Information Flow. There is a rich research literature on techniques that aim to *quantify* information that a program may release, or has released, and then use that quantification as a basis for policy. One question is what measure of information release should be used. Past work largely considers information theoretic measures, including *Bayes vulnerability* [41] and *Bayes risk* [8], *Shannon entropy* [40], and *guessing entropy* [26]. The *g-vulnerability* framework [1] was recently introduced to express measures having richer operational interpretations, and subsumes other measures.

Our work focuses on Bayes Vulnerability, which is related to min entropy. Vulnerability is appealing operationally: As Smith [41] explains, it estimates the risk of the secret being guessed in one try. While challenging to compute, this approach provides meaningful results for non-uniform priors. Work that has focused on other, easier-to-compute metrics, such as Shannon entropy and channel capacity, require deterministic programs and priors that conform to uniform distributions [2, 22, 23, 27, 32]. The work of Klebanov [20] supports computation of both Shannon and Min entropy over deterministic programs with non-uniform priors. The work takes a symbolic execution and program specification approach to QIF. Our use of concolic execution for counting polyhedral constraints is similar to that of Klebanov. However, our language supports probabilistic choice and in addition to concolic execution we also provide a sampling technique and a sound composition. Like Mardziel et al. [25], we are able to compute the worst-case vulnerability, i.e., due to a particular output, rather than a *static* estimate, i.e., as an expectation over all possible outputs. Köpf and Basin [21] originally proposed this idea, and Mardziel et al. were the first to implement it, followed by several others [6, 19, 24].

Köpf and Rybalchenko [22] (KR) also use sampling and concolic execution to statically quantify information leakage. But their approach is quite different from ours. KR uses sampling of a query's inputs in lieu of considering (as we do) all possible outputs, and uses concolic execution with each sample to ultimately compute Shannon entropy, by underapproximation, within a confidence interval. This approach benefits from not having to enumerate outputs, but also requires expensive model counting *for each sample*. By contrast, we use sampling and concolic execution *from the posterior* computed by abstract interpretation, using the results to boost the lower bound on the size/probability mass of the abstraction. Our use of sampling is especially efficient, and the use of concolic execution is completely sound (i.e., it retains 100% confidence in the result). As with the above work, KR requires deterministic programs and uniform priors.

Probabilistic Programming Langauges. A probabilistic program is essentially a lifting of a normal program operating on single values to a program operating on distributions of values. As a result, the program represents a joint distribution over

its variables [18]. As discussed in this paper, quantifying the information released by a query can be done by writing the query in a probabilistic programming language (PPL) and representing the uncertain secret inputs as distributions. Quantifying release generally corresponds to either the maximum likelihood estimation (MLE) problem or the maximum a-posteriori probability (MAP) problem. Not all PPLs support computation of MLE and MAP, but several do.

PPLs based on partial sampling [17, 34] or full enumeration [37] of the state space are unsuitable in our setting: they are either too inefficient or too imprecise. PPLs based on algebraic decision diagrams [9], graphical models [28], and factor graphs [7, 30, 36] translate programs into convenient structures and take advantage of efficient algorithms for their manipulation or inference, in some cases supporting MAP or MLE queries (e.g. [33, 35]). PSI [16] supports exact inference via computation of precise symbolic representations of posterior distributions, and has been used for dynamic policy enforcement [24]. Guarnieri et al. [19] use probabilistic logic programming as the basis for inference; it scales well but only for a class of queries with certain structural limits, and which do not involve numeric relationships.

Our implementation for probabilistic computation and inference differs from the above work in two main ways. Firstly, we are capable of *sound* approximation and hence can trade off precision for performance, while maintaining soundness in terms of a strong security policy. Even when using sampling, we are able to provide precise confidence measures. The second difference is our *compositional* representation of probability distributions, which is based on numerical abstractions: intervals [11], octagons [29], and polyhedra [13]. The posterior can be easily used as the prior for the next query, whereas prior work would have to repeatedly analyze the composition of past queries.

A few other works have also focused on abstract interpretation, or related techniques, for reasoning about probabilistic programs. Monniaux [31] defines an abstract domain for distributions. Smith [42] describes probabilistic abstract interpretation for verification of quantitative program properties. Cousot [14] unifies these and other probabilistic program analysis tools. However, these do not deal with sound distribution conditioning, which is crucial for belief-based information flow analysis. Work by Sankaranarayanan et al [38] uses a combination of techniques from program analysis to reason about distributions (including abstract interpretation), but the representation does not support efficient retrieval of the maximal probability, needed to compute vulnerability.

10 Conclusions

Quantitative information flow is concerned with measuring the knowledge about secret data that is gained by observing the answer to a query. This paper has presented a combination of static analysis using probabilistic abstract interpretation, sampling, and underapproximation via concolic execution to compute high-confidence upper bounds on information flow. Preliminary experimental results are promising and suggest that this approach can operate more precisely

and efficiently than abstract interpretation alone. As next steps, we plan to evaluate the technique more rigorously – including on programs with probabilistic choice. We also plan to integrate static analysis and sampling more closely so as to avoid precision loss at decision points in programs. We also look to extend programs to be able to store random choices in variables, to thereby implement more advanced probabilistic structures.

References

1. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In: Proc. IEEE Computer Security Foundations Symposium (CSF) (2012)
2. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2009)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. International Journal on Software Tools for Tech. Transfer 8(4), 449–466 (2006)
4. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. 72 (Jun 2008)
5. Bagnara, R., Rodríguez-Carbonell, E., Zaffanella, E.: Generation of basic semi-algebraic invariants using convex polyhedra. In: SAS (2005)
6. Besson, F., Bielova, N., Jensen, T.: Browser randomisation against fingerprinting: A quantitative information flow approach. In: NordSec (2014)
7. Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., Van Gael, J.: Measure transformer semantics for bayesian machine learning. In: Proceedings of the European Symposium on Programming (ESOP) (2011)
8. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: On the Bayes risk in information-hiding protocols. Journal of Computer Security 16(5) (2008)
9. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgstroem, J.: Bayesian inference for probabilistic programs via symbolic execution. Tech. Rep. MSR-TR-2012-86, Microsoft Research (2012)
10. Clarkson, M.R., Myers, A.C., Schneider, F.B.: Quantifying information flow with beliefs. Journal of Computer Security 17(5), 655–701 (2009)
11. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming (1976)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL) (1977)
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
14. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Proceedings of the European Symposium on Programming (ESOP) (2012)
15. De Loera, J.A., Haws, D., Hemmecke, R., Huggins, P., Tauzer, J., Yoshida, R.: Latte. https://www.math.ucdavis.edu/~latte/ (2008)
16. Gehr, T., Misailovic, S., Vechev, M.: Psi: Exact symbolic inference for probabilistic programs. In: CAV (2016)

17. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI) (2008)
18. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Conference on the Future of Software Engineering. pp. 167–181. FOSE 2014, ACM, New York, NY, USA (2014)
19. Guarnieri, M., Marinovic, S., Basin, D.: Securing databases from probabilistic inference. In: Proc. IEEE Computer Security Foundations Symposium (CSF) (2017)
20. Klebanov, V.: Precise quantitative information flow analysis—a symbolic approach. Theoretical Computer Science 538, 124–139 (2014)
21. Köpf, B., Basin, D.: An Information-Theoretic Model for Adaptive Side-Channel Attacks. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2007)
22. Köpf, B., Rybalchenko, A.: Approximation and randomization for quantitative information-flow analysis. In: Proceedings of the IEEE Computer Security Foundations Symposium (CSF) (2010)
23. Köpf, B., Rybalchenko, A.: Automation of quantitative information-flow analysis. In: 13th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM), 2013. Lecture Notes in Computer Science, vol. 7938, pp. 1–28. Springer (2013)
24. Kučera, M., Tsankov, P., Gehr, T., Guarnieri, M., Vechev, M.: Synthesis of probabilistic privacy enforcement. In: Proc. ACM Conference on Computer and Communications Security (CCS) (2017)
25. Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. Journal of Computer Security 21, 463–532 (Oct 2013)
26. Massey, J.L.: Guessing and entropy. In: Proc. IEEE Intl. Symposium on Information Theory (ISIT) (1994)
27. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2008)
28. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D.L., Kolobov, A.: Blog: Probabilistic models with unknown objects. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) (2005)
29. Miné, A.: The octagon abstract domain. In: Proceedings of the Working Conference on Reverse Engineering (WCRE) (2001)
30. Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.6 (2014), microsoft Research Cambridge. http://research.microsoft.com/infernet
31. Monniaux, D.: Analyse de programmes probabilistes par interprétation abstraite. Thèse de doctorat, Université Paris IX Dauphine (2001)
32. Mu, C., Clark, D.: An interval-based abstraction for quantifying information flow. Elec. Notes in Theoretical Computer Science 253(3), 119–141 (2009)
33. Narayanan, P., Carette, J., Romano, W., Shan, C.c., Zinkov, R.: Probabilistic inference by program transformation in hakaru (system description). In: Proc. Functional and Logic Programming (2016)
34. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based on sampling functions. ACM Transactions on Programming Languages and Systems (TOPLAS) 31(1), 4:1–4:46 (2008)
35. Pfeffer, A.: Figaro: An object-oriented probabilistic programming language. Tech. rep., Charles River Analytics (2000)

36. Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: Getoor, L., Taskar, B. (eds.) Statistical Relational Learning. MIT Press (2007)
37. Radul, A.: Report on the probabilistic language Scheme. In: Proceedings of the Dynamic Languages Symposium (DLS) (2007)
38. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In: Conference on Programming Language Design and Implementation. PLDI (2013)
39. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/FSE (2005)
40. Shannon, C.: A mathematical theory of communication. Bell System Technical Journal 27 (1948)
41. Smith, G.: On the foundations of quantitative information flow. In: Proc. Conference on Foundations of Software Science and Computation Structures (FoSSaCS) (2009)
42. Smith, M.J.A.: Probabilistic abstract interpretation of imperative programs using truncated normal distributions. Elec. Notes in Theoretical Computer Science (2008)
43. Sweet, I., Trilla, J.M.C., Scherrer, C., Hicks, M., Magill, S.: What's the over/under? probabilistic bounds on information leakage (extended version). CoRR abs/1802.08234 (Feb 2018), https://arxiv.org/abs/1802.08234

All links were last followed on February 23, 2018.

A Query code

The following is the query code of the example developed in Section 2.2. Here, `s_x` and `s_y` represent a ship's secret location. The variables `l1_x`, `l1_y`, `l2_x`, `l2_y`, and `d` are inputs to the query. The first pair represents position L_1, the second pair represents the position L_2, and the last is the distance threshold, set to 4. We assume for the example that L_1 and L_2 have the same y coordinate, and their x coordinates differ by 6 units.

We express the query in the language of Figure 4 basically as follows:

```
d_l1 := |s_x - l1_x| + |s_y - l1_y|;
d_l2 := |s_x - l2_x| + |s_y - l2_y|;
if (d_l1 <= d || d_l2 <= d) then
  out := true // assume this result
else
  out := false
```

The variable `out` is the result of the query. We simplify the code by assuming the absolute value function is built-in; we can implement this with a simple conditional. We run this query probabilistically under the assumption that `s_x` and `s_y` are uniformly distributed within the range given in Figure 1. We then condition the output on the assumption that `out` = `true`. When using intervals as the baseline of probabilistic polyhedra, this produces the result given in the upper right of Figure 3(b); when using convex polyhedra, the result is shown in the lower right of the figure. The use of sampling and concolic execution to augment the former is shown via arrows between the two.

B Formal semantics

Here we defined the probabilistic semantics for the programming language given in Figure 4. The semantics of statement S, written $[\![S]\!]$, is a function of the form $\mathbf{Dist} \to \mathbf{Dist}$, i.e., it is a function from distributions of states to distributions of states. We write $[\![S]\!]\delta = \delta'$ to say that the semantics of S maps input distribution δ to output distribution δ'.

Figure 7 gives this denotational semantics along with definitions of relevant auxiliary operations. We write $[\![E]\!]\sigma$ to denote the (integer) result of evaluating expression E in σ, and $[\![B]\!]\sigma$ to denote the truth or falsehood of B in σ. The variables of a state σ, written $\mathit{domain}(\sigma)$, is defined by $\mathit{domain}(\sigma)$; sometimes we will refer to this set as just the *domain* of σ. We will also use the this notation for distributions; $\mathit{domain}(\delta) \stackrel{\text{def}}{=} \mathit{domain}(\mathit{domain}(\delta))$. We write lfp as the least fixed-point operator. The notation $\sum_{x\,:\,\phi} \rho$ can be read ρ *is the sum over all* x *such that formula* ϕ *is satisfied* (where x is bound in ρ and ϕ).

This semantics is standard. See Clarkson et al. [10] or Mardziel et al [25] for detailed explanations.

$$
\begin{aligned}
[\![\mathsf{skip}]\!]\delta &= \delta \\
[\![x \;:=\; E]\!]\delta &= \delta\,[x \to E] \\
[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]\delta &= [\![S_1]\!](\delta \wedge B) + [\![S_2]\!](\delta \wedge \neg B) \\
[\![\mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]\delta &= [\![S_1]\!](q \cdot \delta) + [\![S_2]\!]((1-q) \cdot \delta) \\
[\![S_1\ ;\ S_2]\!]\delta &= [\![S_2]\!]([\![S_1]\!]\delta) \\
[\![\mathsf{while}\ B\ \mathsf{do}\ S]\!] &= \mathrm{lfp}\,[\lambda f :\ \mathbf{Dist} \to \mathbf{Dist}.\ \lambda\delta. \\
&\qquad\qquad f\,([\![S]\!](\delta \wedge B)) + (\delta \wedge \neg B)]
\end{aligned}
$$

where

$$
\begin{array}{ll}
\delta\,[x \to E] & \stackrel{\text{def}}{=} \lambda\sigma.\ \sum_{\tau\,:\,\tau[x \to [\![E]\!]\tau] = \sigma} \delta(\tau) \\
\delta_1 + \delta_2 & \stackrel{\text{def}}{=} \lambda\sigma.\ \delta_1(\sigma) + \delta_2(\sigma) \\
\delta \wedge B & \stackrel{\text{def}}{=} \lambda\sigma.\ \mathbf{if}\ [\![B]\!]\sigma\ \mathbf{then}\ \delta(\sigma)\ \mathbf{else}\ 0 \\
p \cdot \delta & \stackrel{\text{def}}{=} \lambda\sigma.\ p \cdot \delta(\sigma) \\
\|\delta\| & \stackrel{\text{def}}{=} \sum_\sigma \delta(\sigma) \\
\mathrm{normal}(\delta) & \stackrel{\text{def}}{=} \frac{1}{\|\delta\|} \cdot \delta \\
\delta|B & \stackrel{\text{def}}{=} \mathrm{normal}(\delta \wedge B) \\
\delta_1 \times \delta_2 & \stackrel{\text{def}}{=} \lambda(\sigma_1, \sigma_2).\ \delta_1(\sigma_1) \cdot \delta_2(\sigma_2) \\
\dot{\sigma} & \stackrel{\text{def}}{=} \lambda\sigma_0.\ \mathbf{if}\ \sigma = \sigma_0\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
\sigma \downharpoonright V & \stackrel{\text{def}}{=} \lambda x \in \mathbf{Var}_V.\ \sigma(x) \\
\delta \downharpoonright V & \stackrel{\text{def}}{=} \lambda\sigma_V \in \mathbf{State}_V.\ \sum_{\tau\,:\,\tau\downharpoonright V = \sigma_V} \delta(\tau) \\
\mathrm{f}_x(\delta) & \stackrel{\text{def}}{=} \delta \downharpoonright (\mathit{domain}(\delta) - \{x\}) \\
\mathit{support}(\delta) & \stackrel{\text{def}}{=} \{\sigma\ :\ \delta(\sigma) > 0\}
\end{array}
$$

Fig. 7. Distribution semantics