Anosy: Approximated Knowledge Synthesis with Refinement Types for Declassification

Anonymous Author(s)

Abstract

Non-interference is a popular way to enforce confidentiality of sensitive data. However, declassification of sensitive information is often needed in realistic applications but breaks non-interference. We present Anosy, an approximate knowledge synthesizer for quantitative declassification policies. Anosy uses refinement types to automatically construct machine checked over- and under-approximations of attacker knowledge for boolean queries on multi-integer secrets. It also provides an AnosyT monad to track the attacker knowledge over multiple declassification queries, and checks for violations against the user-specified policies on information flow control applications. We implemented a prototype of Anosy and showed that it is precise and permissive: up to 14 declassification queries were permitted before policy violation using the powersets of interval domain.

1 Introduction

Information flow control (IFC) [33] systems protect the confidentiality of sensitive data during program execution. They do so by enforcing a property called non-interference which ensures the absence of leaks of secret information (say, a user location) through public observations (say, information being sent to the network socket).

Real-world programs, however, often need to reveal information about sensitive data. For instance, a location based web application needs to suggest restaurants or friends that are nearby the `Secret` user location. Such computations, which leak information about the `Secret` location, would be prevented by IFC systems that enforce non-interference. To support them, IFC systems provide *declassification* statements [34] that can be used to weaken non-interference by allowing the selective disclosure of some `Secret` information.

Declassification statements, however, are typically part of an application's trusted computing base and developers are responsible for properly declassifying information. In particular, mistakes in declassification statements can easily compromise a system's security because declassified information bypasses standard IFC checks. Instead of trusting the developer to correctly declassify information, an alternative approach is to enforce *declassification policies* [7] that restrict the use of declassification statements.

In this paper, we present Anosy, a framework for enforcing *declassification policies* on IFC systems where policies regulate *what* information can be declassified [34] by limiting the amount of information an attacker could learn from the declassification statements. Specifically, declassification policies are expressed as constraints over *knowledge* [2], which semantically characterizes the set of secrets an attacker considers possible given the prior declassification statements. To enforce such policies, we develop (1) a novel encoding of knowledge approximations using Liquid Haskell's [43] refinement types which we use to (2) automatically synthesize correct-by-construction knowledge approximations for Haskell queries. We then (3) implement and (4) evaluate a knowledge tracking and policy enforcing declassification function that can easily extend existing IFC monadic systems. Next, we discuss these four contributions in detail.

Verified knowledge approximations. We define a novel encoding for knowledge approximations over abstract domains using Liquid Haskell (§ 4). The novelty of our encoding is that approximation data types are indexed by two predicates that respectively capture the properties of elements inside and outside of the domain. Using these indexes, we encode correctness of over- and under-approximations, without using quantification, permitting SMT-decidable verification. With this encoding, we implement and machine check Haskell approximations of two abstract domains: intervals over multi-dimensional spaces (where each dimension is abstracted using an interval), and powersets on these intervals, that increase the precision of our approximations. This verified knowledge encoding is general and can be used, beyond declassification, also as building block for dynamic [13, 40], probabilistic [14, 19, 24, 39], and quantitative policies [3, 18].

Synthesis of knowledge approximations. We develop a novel approach for automatically synthesizing correct-by-construction posteriors given any prior knowledge and user-specified boolean query over multi-dimensional integer secret values (§ 5). Our approach combines type-based sketching with SMT-based synthesis and is implemented as a Haskell compiler plugin, *i.e.,* it operates at compile-time on Haskell programs. Given a user-defined query, Anosy generates a template where the values of the abstract domain elements are left as *holes* to be filled later with values, combined with the correctness specification encoded as refinement types. It then reduces the high-level correctness property into integer constraints on bounds of the abstract domain elements, and we use an SMT solver to synthesize *optimal* correct-by-construction values. Replacing these values in the sketch, we synthesize Haskell executable programs of

PL'18, January 01–03, 2018, New York, NY, USA
2018.

the approximated knowledge and we use Liquid Haskell to automatically check their correctness.

Enforcing declassification policies. We implement a policy-based declassification function that can be used by any monadic Haskell IFC framework (§ 2, § 3). In this setting, users write declassification policies as Haskell functions that constrain the (approximated) attacker knowledge, whereas declassification queries are written as regular Haskell functions over secret data. At compile time, Anosy synthesizes and verifies the knowledge approximations for all declassification queries. At runtime, declassification is called in the `AnosyT` monad that tracks knowledge over multiple declassification queries and checks, using the synthesized knowledge approximations, whether performing the declassification would lead to violating the user-specified policy. Importantly, `AnosyT` is defined as a monad transformer, thus can be staged on top of existing IFC monads like LIO [38] and STORM [20].

Evaluation. We evaluated precision and running time of Anosy using two benchmarks (§ 6). First, we compared with `Prob`'s [24] benchmark suite to conclude that Anosy is slower but more precise. Second, in an attempt to declassify 50 consecutive queries, a policy violation was detected after maximum 7 queries with the interval abstract domain and after 14 queries with the more precise powerset domain.

2 Overview

We start by motivating the need for declassification policies (§ 2.1): repeated downgrades can weaken non-interference until leaking the secret is allowed. Next, we present how the knowledge revealed by queries can be computed (§ 2.2). Finally (§ 2.3), we describe how Anosy synthesizes correct-by-construction knowledge, by combining refinement types, SMT-based synthesis, and metaprogramming.

2.1 Motivation: Bounded Downgrades

Secure Monads. IFC systems, *e.g.,* LIO [38] and LWeb [27], define a *secure* monad to ensure that security policies are enforced over sensitive data, like a user's physical location. For instance here, we define the data type `UserLoc` to capture the user location as its `x` and `y` coordinates.

```
data UserLoc = UserLoc {x :: Int, y :: Int}
```

A `Secure` monad will return such a location wrapped in a protected "box" to ensure that only code with sufficient privileges can inspect it. For example, a function that gets the user's location will return a protected value:

```
getUserLoc :: User → Secure (Protected UserLoc)
```

In the `LIO` monad, for example, data are protected by a security label data type, and the monad ensures, based on the application, that only the intended agents can observe (or unlabel) the user's exact location.

Queries. In the following, *query* is any boolean function over *secret* values. As an example, we consider the user location to be the secret value and the `nearby` function below checks proximity to this secret value from `(x_org, y_org)`.

```
type 𝒮 = UserLoc

nearby :: (Int, Int) → 𝒮 → Bool
nearby (x_org, y_org) (UserLoc x y)
  = abs (x - x_org) + abs (y - y_org) ≤ 100
  where abs i = if i < 0 then -i else i
```

The `nearby` query is using Manhattan distance to check if a user is located within `100` units of the input origin location.

Downgrades. Even though locations protected by the `Secure` monad cannot be inspected by unprivileged code, in practice many applications need to allow selective leaks of secret information to unprivileged code. For instance, many web applications need to check location of users to provide usable information, such as restaurant, friend, or dating suggestions that are physically nearby the user.

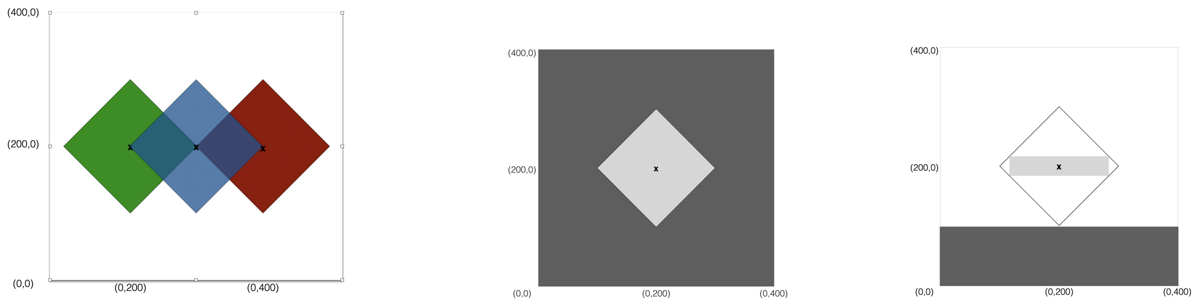
The `showAdNear` function below shows a restaurant advertisement to the user only if they are nearby. To do so, the function uses `downgrade` (from the `Secure` monad) to downgrade (to `public`) the result of the `nearby` check over the protected user location.

```
downgrade  :: (Protected 𝒮) → (𝒮 → Bool)
           → Secure Bool

showAd     :: User → Restaurant → Secure ()
showAdNear :: User → Restaurant → Secure ()
showAdNear user res = do
  ul     ← getUserLoc user
  isNear ← downgrade ul (nearby (res_loc res))
  if isNear then showAd user res else return ()
```

Downgrades are a common feature of real-world IFC systems. For example, in LIO downgrades happen with the `unlabelTCB` trusted codebase function, which is exposed to the application developers. At the same time, downgraded information bypasses security checks by design. In the code above, `isNear` is unprotected and can now be leaked to an attacker. Therefore, declassification statements need to be correctly placed to avoid unintended leaks of information that would bypass IFC enforcement.

Declassification knowledge. To semantically characterize the information declassified by `downgrade`s, we use the notion of *attacker knowledge* [2], *i.e.,* the set of secrets that are consistent with an attacker's observations, where attackers can observe the results of `downgrade`. That is, we consider the worst-case scenario where any declassified information is *always* leaked to an attacker. This knowledge can be refined by consecutively downgrading queries and ultimately can reveal the exact value of the secret. For example, below,



(a) Posteriors on (200,200), (300,200), and (400,200).

(b) Indistinguishability Sets

(c) Under-Approximation

Figure 1. Posteriors, Indistinguishability Sets and their Approximations with respect to `nearby` query.

a piece of code downgrades two queries asking if the user is located nearby to both the origins `(200,200)` and `(400,200)` to infer if the exact user location is `(300,200)`.

```
secret ← getUserLoc
kn1 ← downgradeUs secret (nearby (200,200))
kn2 ← downgradeUs secret (nearby (400,200))
-- if kn1 ∧ kn2 then secret = (300,200)
```

We call *posterior* the knowledge obtained after executing a query. Consider again the code above. If `nearby (200,200)` is true, the knowledge after the first downgrade statement is the green region of Figure 1a. Using this information as *prior* knowledge for the second downgraded query, which asks `nearby (200,400)`, might result in a knowledge containing only the user location `(200,300)`, *i.e.,* the intersection of the green and red posterior knowledge regions.

Quantitative Policies. A *quantitative policy* is a predicate on knowledge which, for instance, ensures that the accumulated knowledge is not specific enough, *i.e.,* the secret cannot be revealed. As an example, the `qpolicy` below states that the knowledge should contain at least `100` values.

```
qpolicy dom = size dom > 100
```

This policy will allow declassifying `nearby (200,200)` and `nearby (300,200)`, since the intersections of the green and blue regions in Figure 1a contain at least 100 potential locations, but not `nearby (400,200)` since the resulting knowledge contains exactly one secret.

Bounded Downgrade. We define a bounded downgrade operator that allows the computation of queries on secret data, while enforcing quantitative policies. For example, the operator tracks declassification knowledge during the execution and allows downgrading the `nearby (200,200)` and `nearby (200,300)` queries, but terminates with an error on the sequence of `nearby (200,200)` and `nearby (200,400)`.

The downgrade operation is the method of the `AnosyT` monad (§ 3) which is defined as a state monad transformer. As a state monad, it preserves the protected secret, the quantitative policy, and the prior declassification knowledge. To `downgrade` a new query, the monad checks if the posterior knowledge of this query satisfies the policy. If not, it terminates with a policy violation error. Otherwise, it updates the knowledge to the posterior and returns the query result. Since `AnosyT` is also a monad transformer, it can be combined with existing security monads, which provide the underlying IFC enforcement mechanism, to enrich them with extra quantitative guarantees on the inevitable downgrades.

2.2 Approximating knowledge from queries

Precisely computing, representing, and checking quantitative policies over a (potentially infinite) knowledge requires reasoning about all points in the input space, which is an uncomputable task in general. So, we use abstract domains (here intervals [10]) to approximate knowledge.

Indistinguishability sets. The proximity query `nearby (200,200)` partitions the space of secret locations into two partitions (for the two possible responses: `True` and `False`), called *indistinguishability sets* (ind. sets), *i.e.,* all secrets in each partition produce the same result for the query. Figure 1b depicts the two ind. sets for our query. The inner diamond—depicted in light gray—is the ind. set for the result `True`, *i.e.,* all its elements respond `True` to the query. In contrast, the outer region—depicted in dark gray—is the ind. set for `False`. Figure 1c depicts the under-approximated (*i.e.,* subset) ind. sets for the query as defined by the `under_indset` zero argument function below:

```
data 𝒜Int = 𝒜Int {lower :: Int, upper :: Int}
data 𝒜 = 𝒜 [𝒜Int]

under_indset :: (𝒜, 𝒜)
under_indset = (𝒜 [𝒜Int 121 279, 𝒜Int 179 221],
                𝒜 [𝒜Int   0 400, 𝒜Int   0  99])
```

The data `𝒜Int` abstracts integers as intervals between a lower and an upper value. `𝒜` is our abstract knowledge data type that is defined as a list of abstract integers, which can be used to abstract data with any number of integer fields. The `under_indset` is a tuple, where the first element corresponds to the `True` response and the second element to the `False` response. It says all secrets in $x \in [121, 279]$ and $y \in [179, 221]$ evaluate to `True` for the query and all secrets in $x \in [0, 400]$ and $y \in [0, 99]$ evaluate to `False`.

Knowledge under-approximation. We use ind. sets to compute the *posterior* knowledge after the query, *i.e.,* the set of secrets considered possible after observing the query result. To do so, we simply take the intersection ∩ of the prior knowledge with the ind. set associated with the query [2, 3]. If the intersection happens with the exact ind. sets, then we derive the exact posterior. For our example, we intersect with the under-approximate ind. set to produce an under-approximation of the posterior knowledge *i.e.,* an under-approximation of the information learned when observing the query result.

```
underapprox :: 𝒜 →  (𝒜, 𝒜)
underapprox p = (p ∩ trueInd, p ∩ falseInd)
where (trueInd, falseInd) = under_indset
```

The intersection ∩ refers to the set-theoretic intersection of two domains. We formally define these operations in § 4.

2.3 Verification and Correct-by-Construction Synthesis of Knowledge

Our goal is to generate a knowledge approximation for each downgraded query, which as shown by our `nearby` example is a strenuous and error prone process. To automate this process we use refinement types, metaprogramming, and SMT-based synthesis to automatically generate correct-by-construction knowledge approximations of queries in four steps. First, for each query we generate a refinement type specification that denotes knowledge approximation. Next, we use metaprogramming to generate a template, *i.e.,* a function definition with holes that computes the knowledge. Then, we use an SMT to fill in the integer value holes in the template. Finally, we use Liquid Haskell's refinement type checker to verify that our synthesized knowledge indeed satisfies its specification.

Here, we explain a simplified version of these steps for our `nearby (200,200)` example query.

Step I: Refinement Type Specifications. We use abstract refinement types to index abstract domains with a predicate that all its elements should satisfy (§ 4). For example, `𝒜<{\l → 0 < l}>` denotes the abstract domain whose elements are positive values. Using this abstraction, we specify the ind. set and knowledge approximations as follows:

```
under_indset :: (𝒜<{\l → query l}>,
                 𝒜<{\l → ¬ query l}>)


underapprox :: p: 𝒜
             → (𝒜 <{\x →   query x ∧ (x ∈ p)}>,
                𝒜 <{\x → ¬ query x ∧ (x ∈ p)}>)
```

The `under_indset` returns a tuple of abstract domains. The first abstract domain can only contain elements that satisfy the query and the second that falsify it. The `underapprox` is further refined to contain only elements that originally existed in the prior knowledge.

Step II: Template Function Generation. Using syntax directed metaprogramming we define `underapprox` as in § 2.2 to be the intersection of the ind. set and the prior knowledge. For the definition of the ind. set we rely on the secret type to be abstracted to generate a template with integer value holes. For our running example, and since the `UserLoc` contains two integer fields, the template of `under_indset` is the following program sketch [37]:

$$\texttt{under\_indset = (}\mathcal{A}\ \texttt{[}\mathcal{A}\texttt{Int}\ \texttt{l}_{t1}\ \texttt{u}_{t1}\texttt{,}\ \mathcal{A}\texttt{Int}\ \texttt{l}_{t2}\ \texttt{u}_{t2}\texttt{],}\ \ \mathcal{A}\ \texttt{[}\mathcal{A}\texttt{Int}\ \texttt{l}_{f1}\ \texttt{u}_{f1}\texttt{,}\ \mathcal{A}\texttt{Int}\ \texttt{l}_{f2}\ \texttt{u}_{f2}\texttt{])}$$

Step III: SMT-Based Synthesis. Finally, we combine the refinement type with the program sketch to generate, using an SMT, solutions of the unknown integers (§ 5). In our example, we will get the below two constraints:

$$\forall x, y.\ l_{t1} \le x \le u_{t1}\ \wedge\ l_{t2} \le y \le u_{t2} \implies query(x, y) \qquad \text{(Under-approx, True)}$$

$$\forall x, y.\ l_{f1} \le x \le u_{f1}\ \wedge\ l_{f2} \le y \le u_{f2} \implies \neg query(x, y) \qquad \text{(Under-approx, False)}$$

These constraints have multiple correct solutions, but we would prefer the tightest bounds wherever possible. This translates to generating the maximal, most precise domain when under-approximating. We use Z3 [5] as the SMT solver of choice because it supports these optimization directives to maximize $u_1 - l_1$ and $l_2 - u_2$ together, for both the true and false cases. Finally, we use these solutions to fill in the templates and derive complete programs.

Step IV: Knowledge Verification. We use Liquid Haskell to verify the synthesized result. To achieve this step, we implemented (§ 4) verified abstract domains for intervals and their powersets that, as shown in our evaluation § 6, greatly increase the precision of the abstractions. These implementations are independent of the synthesis step and can be used to verify user-written, knowledge approximations.

3 Bounded Downgrade

Here we present the bounded downgrade operation, first by an example that showcases how downgrades that violate the quantitative declassification policy are rejected, next by providing its exact implementation, and finally by showing correctness of policy enforcement.

Bounded Downgrade by Example. The bounded downgrade function checks, before downgrading a query using the underlying `Secure` monad, that the approximation of the revealed knowledge satisfies the quantitative policy. To do so, it preserves a state that maps each secret that has been involved in downgrading operations to its current knowledge. As an example, below we present how the knowledge is updated to prevent the example from § 2.1.

```
secret ← lift getUserLoc
-- secret  = Protected (UserLoc 300 200)
-- secrets = []
r1 ← downgrade secret "nearby (200,200)"
-- secrets = [(secret, post1 = {121...279,179...221})], |post1| = 6837
r2 ← downgrade secret "nearby (300,200)"
-- secrets = [(secret, post2 = {221...279,179...221})], |post2| = 2537
r3 ← downgrade secret "nearby (400,200)"
-- secrets = [(secret, post3 = {∅        , 179 ... 221})], |post3| = 0
-- Policy Violation Error
```

The user location is taken by lifting the `getUserLoc` function of the underlying monad (any computation of the underlying monad can be lifted). Assume that the user is located at `(300,200)`. Originally, there is no prior knowledge for this secret (and protected) location, *i.e.,* the `secrets` map associating secrets to knowledge approximations is empty. After downgrading the `nearby (200,200)` query (which as we will explain next, is passed to `downgrade` as a string) we get the posterior `post1` with size `6837`. Since this size is greater than `100`, the `qpolicy` (defined in § 2.1) is satisfied and the result of the query (here true) is returned by the bounded downgrade. Similarly, downgrade of the `nearby (300,200)` query refines the posterior to size `2537`. But, when downgrading the `nearby (400,200)` query the posterior size becomes zero, thus our system will refuse to perform the query (and downgrading its result) and return a policy violation error, instead of risking the leak of the secret.

Definition of Bounded Downgrade. Figure 2 presents the definition of the bounded `downgrade` function. It takes as input a protected secret, which should be able to get `unprotected` by an instance of the `Unprotectable` class, a string that uniquely determines the query to be executed, and returns a boolean value in the `AnosyT` state monad transformer [22]. As discussed in § 2.1, we used a transformer to stage our downgrade on top of an existing secure monad.

The state of Anosy `AState` contains the quantitative policy, the map `secrets` of secret values to their current knowledge, and the map `queries` that maps strings that represent queries to query information `QInfo` that, in turn, contain both the query itself and an under-approximation function (like the synthesized `underapprox`) that given the prior knowledge approximates the posterior, after the query is executed. Even though tracking of multiple secrets is permitted, we require all the secrets and abstractions to have the same type; this limitation can be lifted using heterogeneous collections [17].

Having access to this state, `downgrade` will throw an error if it cannot find the query information of the string input, since it has no way to generate posterior knowledge[1]. Then, it will compute the posterior and throw an error if it violates the quantitative policy. Otherwise, it will update the posterior of the secret and return the result of the query.

```
type AnosyT a s m = StateT (AState a s) m

data AState a s = AState {
  policy   :: a → Bool,
  secrets  :: Map s a,
  queries  :: Map String (QInfo a s)}

data QInfo a s = QInfo {
  query  :: s → Bool,
  approx :: p: a
         → (a <{\x →   query x ∧ (x ∈ p)}>,
            a <{\x → ¬ query x ∧ (x ∈ p)}>)}

class Unprotectable p where
  unprotect :: p t → t

downgrade :: ( Monad m, Unprotectable protected
             , AbstractDomain a s) -- Defined in § 4.1
          ⇒ protected s
          → String -- (s → Bool)
          → AnosyT a s m Bool
downgrade secret' qName = do
  st ← get
  let qinfo  = lookup qName (queries st)
  if isJust qinfo then do
    let secret = unprotect secret'
    let prior  = fromMaybe ⊤
                 $ lookup secret (secrets st)
    let (QInfo query approx) = fromJust qinfo
    let (postT, postF) = approx prior
    if policy st postT ∧ policy st postF then do
      let response = query secret
      let posterior = if response then postT
                      else postF
      modify $ \st → st {secrets =
        insert secret posterior (secrets st)}
      return $ response
    else throwError "Policy Violation"
  else throwError ("Can't downgrade " ++ qName)
```

Figure 2. Implementation of bounded `downgrade`.

Correctness: Policy Enforcement. Suppose a secret `s` that has been downgraded n times by the queries $\texttt{query}_1, \ldots, \texttt{query}_n$. After each downgrade, the knowledge is refined. So, starting from the top knowledge ($\mathcal{K}_0 \doteq \top$), after n queries, the knowledge evolves as follows: $\mathcal{K}_0 \subseteq \mathcal{K}_1 \subseteq \cdots \subseteq \mathcal{K}_i \subseteq \cdots \subseteq \mathcal{K}_n$, where $\mathcal{K}_i = \mathcal{K}_{i-1} \cap \{x \mid \text{query}_i\ x = \text{query}_i\ \text{s}\}$.

[1] On-the-fly synthesis albeit possible would be very expensive.

We can show that for each i-th downgrade of the secret s, there exists a posterior $\mathcal{P}_i$ so that (s,$\mathcal{P}_i$) is in the secrets map and also $\mathcal{P}_i$ is an under-approximation of the knowledge $\mathcal{K}_i$, that is $\mathcal{P}_i \subseteq \mathcal{K}_i$. The proof goes by induction on i, assuming that the attacker and the downgrade implementation start from the same ⊤ knowledge, and the inductive step relies on the specification of the approx function and the way downgrades modify secrets, *i.e.,* using postT or postF depending on the response of the query.

Thus if our quantitative policy enforces a lower bound on the size of the leaked knowledge, (*e.g.,* qpolicy dom = size dom > k) it is correctly enforced by downgrade: since $\mathcal{P}_i \subseteq \mathcal{K}_i$, then qpolicy $\mathcal{P}_i$ implies qpolicy $\mathcal{K}_i$ at each stage of the execution. Note that for correctness of policy enforcement, the policy should be an increasing function in the size of the input for underapproximations. The exact definition of such a policy domain specific language is left as a future work. Further, even though our implementation can trace knowledge overapproximations, we have not yet studied applications or policy enforcement of this case. Last but not least, it is important that the policy is checked irrespective of the query result, *i.e.,* on both postT and postF, to prevent potential leaks due to the security decision.

4 Refinement Types Encoding

We saw that our bounded downgrade function is correct, if each query is coupled with a function approx that correctly computes the underapproximation of posterior knowledge. Here, we show how refinement types can specify correctness of approx, in a way that permits decidable refinement type checking. First (§ 4.1), we define the interface of abstract domains as a refined type class that in § 4.2 we use to specify the abstractions of ind. sets and knowledge. Next, we present two concrete instances of our abstract domains: intervals (§ 4.3) and powersets of intervals (§ 4.4).

4.1 Abstract Domains

Figure 3 shows the AbstractDomain a s refined type class interface stating that a can abstract, *i.e.,* represent a set of values of, s. For example, an instance **instance** $\mathcal{A}_I$ UserLoc states that the data type $\mathcal{A}_I$ (that we will define in § 4.3) abstracts UserLoc (of § 2.1). The interface contains method definitions and class laws, and when required the abstract domain is indexed by abstract refinements.

Class Methods. The class contains six, standard, set-theoretic methods. Top (⊤) and bottom (⊥), respectively represent the full and empty domains. Member $c \in d$ tests if the concrete value c is included in the abstract domain d. Subset $d_1 \subseteq d_2$ tests if the abstract domain d_1 is fully included in the abstract domain d_2. Intersect $d_1 \cap d_2$ computes an abstract domain that includes all the concrete values that are included in both its input domains. Finally, size s computes the number of concrete values represented by an abstract domain.

```
class AbstractDomain a s where
  ⊤    :: a<{\_ →  True, \_ → False}>
  ⊥    :: a<{\_ → False, \_ →  True}>
  ∈    :: s → a → Bool
  ⊆    :: a → a → Bool
  ∩    :: d1:a<p1, n1> → d2:a<p1, n1>
       → {d3:a<p1∧p2, n1∨n2> | d1 ⊆ d3 ∧ d2 ⊆ d3}
  size :: a → {i:Int | 0 ≤ i}
  -- class laws
  sizeLaw   :: d1:a → {d2:a | d1 ⊆ d2}
            → {size d1 ≤ size d2}
  subsetLaw :: c:s → d1:a → {d2:a | d1 ⊆ d2}
            → {c ∈ d1 ⇒ c ∈ d2}
```

Figure 3. Abstraction Domain Type Class

Class Laws. We use refinement types to specify two class laws that should be satisfied by the ⊆ and size methods. sizeLaw states that if d_1 is a subset of d_2, then the size of d_1 should less or equal to the size of d_2. subsetLaw states that if d_1 is a subset of d_2 then, any concrete value in d_1 is also in d_2. These methods have no computational meaning (*i.e.,* they return unit) but should be instantiated by proof terms that satisfy the denoted laws. Even though we could have expressed more set-theoretic properties as laws, these two were the ones required to verify our applications.

Abstract Indexes. In the types of top, bottom, and intersection, the type a is indexed by two predicates p and n (s → Bool). The positive predicate p describes properties of concrete values that are members of the abstract domain. Likewise, the negative predicate n describes properties of the values that do not belong to the abstract domain. Intuitively, the meaning of these predicates is the following:

```
a<p,n> ~ {d:a | ∀x. x∈d ⇒ p x ∧  ∀x. x∉d ⇒ n x}
```

Yet, the right-hand side definition is using quantifiers which lead to undecidable verification. Instead, we used abstract refinements [42] and the left-hand side encoding, to ensure decidable verification.

The specification of the full domain ⊤ states that the positive predicate is True, *i.e.,* satisfied by all elements of the domain, and the negative False, *i.e.,* no elements are outside of the domain. Similarly, the empty domain ⊥ has a False positive predicate, *i.e.,* no elements are in the domain, and True negative predicate, *i.e.,* all elements can be outside the domain. Finally, the type signature for intersect $d_1 \cap d_2$ returns a domain d_3 whose positive predicate indicates it includes elements included in d_1 and d_2 *i.e.,* $p_1 \wedge p_2$. The negative predicate indicates points excluded from d_3 are points excluded from either d_1 or d_2, *i.e.,* $n_1 \vee n_2$. The refinement on d_3 ensure that d_3 is a subset ⊆ of both d_1 and d_2. For abstract

```
query :: s → Bool

under_indset :: (a<{\x →   query x,true}>,
                 a<{\x → ¬ query x,true}>)
over_indset  :: (a<{true,\x → ¬ query x}>,
                 a<{true,\x →   query x}>)

underapprox :: p:a →
  (a<{\x →   query x ∧ (x ∈ p), true}>,
   a<{\x → ¬ query x ∧ (x ∈ p), true}>)
underapprox p = (dT ∩ p,dF ∩ p)
  where (dT,dF) = over_indset
overapprox  :: p:a →
  (a<{true, \x → ¬ query x ∨ (x ∉ p)}>,
   a<{true, \x →   query x ∨ (x ∉ p)}>)
overapprox p = (dT ∩ p,dF ∩ p)
  where (dT,dF) = over_indset
```

Figure 4. Specifications of Approximations for concrete *a* and *s* that instantiate `AbstractDomain`.

types in which these two predicates are omitted, the `\_ → True` predicate is assumed, which we will from now on abbreviate as `true` and imposes no verification constraints.

4.2 Approximations of ind. sets and knowledge

In Figure 4, we use the positive and negative abstract indexes to encode the specifications of over- and under-approximations for ind. sets and knowledge. We assume concrete types for *a* and *s* with an **instance** *a s* and a query on the secret. (In the previous sections for simplicity, we omitted the negative predicates and overapproximations.)

Approximations of ind. sets. A query's ind. sets is a tuple whose first element contains all the secrets satisfying the query and its second all the secrets falsifying the query.

The specification of the ind. set `under_indset` says the first domain only includes secrets for which the `query` is `True`, and the second domain only includes secrets for which the `query` is `False` (the positive predicates). The negative predicates do not impose any constraints on the elements that do not belong to the domain. This means the domains can exclude any number of secrets, as long as the secrets that are included are correct, *i.e.*, it is an under-approximation.

Dually, the over-approximation `over_indset` sets the negative predicate to exclude all points for which the `query` evaluates to `False` for the domain corresponding to `True` response, and the second domain (corresponding to response `False`) excludes all points that evaluate to `False` with `query`. The positive predicates are just true. The domains can include any number of secrets as long as they are not leaving out any secrets that are correct, *i.e.,* it is an over-approximation.

Approximations of knowledge. By combining the prior knowledge of the attacker with the ind. set for the query, we derive an approximation of the attacker's knowledge after they observe the query. Figure 4 shows the specifications for the knowledge under-approximation `underapprox` and the over-approximation `overapprox`. `underapprox` is similar to the type of `under_indset`, except the positive predicate is strengthened to express that all the elements of the domain should also belong to the prior knowledge `p`. Similarly, `overapprox` specifies that the elements that do not belong in the posterior knowledge, should neither be in the prior nor the ind. set. Each approximation is implemented by a pairwise intersection with the respective ind. sets and can be verified because of the precise type we gave to intersection.

Precision. The refinement types ensure our definitions are correct, but they do not reason about the precision of the abstract domains. For example, the bottom ⊥ and top ⊤ are vacuously correct solutions for under- and over-approximations, respectively. But, these domains are of little use as ind. sets, since they ignore all the query information. It is unclear if precision of an abstract domain can be encoded using refinement types, instead, we evaluate it empirically in § 6.

4.3 The Interval Abstract Domain

Next we define $\mathcal{A}_I$, the interval abstract domain that can abstract any secret type $\mathcal{S}$, constructed as a product of integers (like the `UserLoc` of § 2) or types that can be encoded to integers (*e.g.*, booleans or enums). $\mathcal{A}_I$ is defined as follows:

```
-- 𝒮 = Int × Int × ...
data 𝒜Int = 𝒜Int {lower :: Int, upper :: Int}
type Proof p x = {v:𝒮<p> | v = x }

data 𝒜_I <p::𝒮 → Bool, n::𝒮 → Bool>
  = 𝒜_I { dom :: [𝒜Int]
        , pos :: x:{𝒮| x ∈ dom } → Proof p x
        , neg :: x:{𝒮| x ∉ dom } → Proof n x }
  | ⊤_I { pos :: x:𝒮 → Proof p x }
  | ⊥_I { neg :: x:𝒮 → Proof n x }
```

$\mathcal{A}_I$ has three constructors. $\top_I$ and $\bot_I$ respectively denote the complete and empty domains. $\mathcal{A}_I$ represents the domain of any n-dimensional intervals, where `n` is the length of `dom`. An interval `𝒜Int` represents integers between `lower` and `upper`. For a secret $s = s_1 \times s_2 \times \ldots s_n$, an $\mathcal{A}_I$ represents each s_i by the `i`th element of its `dom` ($s_i \in$ `(dom!i)`) in the `n` dimensional space. For example, `domEx = [(𝒜Int 188 212), (𝒜Int 112 288)]` is the rectangle of $x \in [188, 212], y \in [112, 288]$ in the two dimensional space of `UserLoc`.

Proof Terms. The `pos` and `neg` components in the $\mathcal{A}_I$ definition are proof terms that give meaning to the positive `p` and negative `n` abstract refinements. The complete domain $\top_I$ contains the proof field `pos` that states that every secret *s* should satisfy the positive predicate p (*i.e.,* x: $\mathcal{S}$ → Proof p x), and the empty domain contains only the proof neg for the negative predicate n. Due to syntactic restrictions that abstract refinements can only be attached to a type for SMT-decidable verification [42], the proof terms terms are encoded as functions that return the secret, while providing evidence that the respective predicates is inhabited by possible secrets. In $\mathcal{A}_I$ this is encoded by setting preconditions to the proof terms: the type of the pos field states that each s that belongs to dom should satisfy p, while the neg field states that each x that does not belong to dom should satisfy n.

When an $\mathcal{A}_I$ is constructed via its data constructors, the proof terms should be instantiated by explicit proof functions. For example, below we show that the domEx (described above) only represents elements that are nearby (200,200).

```
example :: 𝒜_I <{\s → nearby (200,200) s, true}>
example = 𝒜_I domEx exPos (\x → x)

exPos :: s:{UserLoc | s ∈ domEx }
      → {o:UserLoc | nearby (200,200) s ∧ o = s}
exPos (UserLoc x y) = UserLoc x y
```

The proof term exPos is an identity function refined to satisfy the pos specification. Once the type signature of exPos is explicitly written, Liquid Haskell is able to automatically verify it. Automatic verification worked for all non-recursive queries, but for more sophisticated properties (*e.g.,* in the definition of the intersection function) we used Liquid Haskell's theorem proving facilities [41] to establish the proof terms. Importantly, when $\mathcal{A}_I$ is used opaquely (*e.g.,* in the approx of Figure 4), the proof terms are automatically verified.

AbstractDomain Instance. We implemented the methods of the AbstractDomain class for the $\mathcal{A}_I$ data type as interval arithmetic functions lifted to n-dimensions. ∈ checks if any secret is between lower and upper for every dimension. ⊆ checks if the intervals representing the first argument is included in the intervals representing the second argument. ∩ computes a new list of intervals to represent the abstract domain, that includes only the common concrete values of the arguments. Size just computes the number of secrets in the domain, which can be interpreted as the volume. Our implementation consists of 360 lines of (Liquid) Haskell code, the vast majority of which constitutes explicit proof terms for pos and neg fields and the class law methods. By design, $\mathcal{A}_I$ uses a list to abstract secrets that are sums of any number of elements, thus this class instance can be reused by an Anosy user to abstract various secret types.

4.4 The Powersets of Intervals Abstract Domain

To address the imprecision of the interval abstract domains, we follow the technique of [4, 30] and define the powerset abstract domain $\mathcal{A}_{\mathbb{P}}$ *i.e.,* a set of interval domains. Similar to intervals, powerset $\mathcal{A}_{\mathbb{P}}$ is also parameterized with the positive and negative predicates:

```
data 𝒜_ℙ <p::𝒮 → Bool, n::𝒮 → Bool> = 𝒜_ℙ {
    dom_i :: [𝒜_I] , dom_o :: [𝒜_I]
  , pos :: x:{𝒮|x ∈ dom_i ∧ x ∉ dom_o} → Proof p x
  , neg :: x:{𝒮|x ∉ dom_i ∨ x ∈ dom_o} → Proof n x}
```

$\mathcal{A}_{\mathbb{P}}$ contains four fields. dom_i is the set (represented as a list) of intervals that are contained *in* the powerset. dom_o is the set of intervals that are *excluded* from the powerset. This representation backed by two lists gives flexibility to define powersets by writing regions that should be included and excluded, without sacrificing generality or correctness (as guaranteed by our proofs). Moreover, this encoding of the powerset makes our synthesis algorithm simpler (§ 5). The proof terms provide the boolean predicates that give semantics to the secrets contained in the powerset, similar to the interval abstract domain (§ 4.3). We do not need a separate top ⊤ and bottom ⊥ for $\mathcal{A}_{\mathbb{P}}$ as they can be represented using $\top_I$ or $\bot_I$ in the pos list.

AbstractDomain Instance. We implemented the methods of the AbstractDomain class for the powerset abstraction in 171 lines of code. A concrete value belongs to (∈) the powerset $\mathcal{A}_{\mathbb{P}}$ if it belongs to any individual interval of the dom_i list but not in any individual interval of the dom_o list. The subset $d_1 \subseteq d_2$ operation checks if all the individual intervals in the inclusion list dom_i of d_1 is a subset of at least one interval in the inclusion list dom_i of d_2, and also that none of the individual intervals in the exclusion list dom_o of d_1 is a subset of any interval in dom_o of d_2. This operation returns True is the first powerset is a subset of the second, but if it returns False it may or may not be powerset. We have not found this to be limiting in practice, as this criteria is sufficient for verification. We plan to improve the accuracy via better algorithms in future work. Intersection $d_1 \cap d_2$ produces a new powerset, whose inclusion list is made of pair-wise intersecting intervals from dom_i of d_1 and dom_i of d_2, and the exclusion interval list is simply the union of all intervals in the individual exclusion lists dom_o of d_1, and dom_o of d_2. Size is computed by taking the sum of size of all intervals in the inclusion list less the size of all intervals in the exclusion list.

5 Synthesis of Optimal Domains

We use synthesis in Anosy to automatically generate ind. sets that satisfy the correctness types of Figure 4 for each query that is downgraded. Our synthesis technique proceeds in three steps: first, Anosy extracts the sketch of the posterior computation (§ 5.1). Second, it translates this to SMT constraints with relevant optimization directives to synthesize the abstract domains (§ 5.2). Finally, the SMT synthesis is iterated to allow synthesis of powersets of any size (§ 5.3).

5.1 Synthesis Sketch

We use syntax-directed synthesis to generate a sketch [37], *i.e.,* a partial program, for the ind. sets functions based on their type specifications of Figure 4. For example, the sketch for `underapprox` would be the following:

```
under_indset = (□::𝒜<{\x →   query x,\_ → True}>,
                □::𝒜<{\x → ¬ query x,\_ → True}>)
```

Following the structure of the type we simply introduce *typed* holes of the form `□::t` for each abstract domain.

5.2 Synth: SMT-based Synthesis of Intervals

We define the procedure Synth that given a typed hole of an abstract domain, the number of fields in the secret n, and the kind of approximation (over or under), it returns a solution, *i.e.,* an abstract domain that satisfies the hole type. As an example, consider the first hole of the `under_indset` as an interval domain.

```
□  :: 𝒜_I <{\x → query x,\_ → True}>
□   = 𝒜_I dom pos neg
dom = [𝒜Int l_1 u_1, ..., 𝒜Int l_n u_n]
```

The solution is using the $\mathcal{A}_I$ applied to the domain list `dom` and the `pos` and `neg` proof terms. The `dom` is a list of `𝒜Int` that contain unknown integers as lower and upper bounds, while the length n of the list is defined to be the number of (protected) fields of the secret data type. The proof terms for our (non recursive) queries follow concrete patterns (as the example of § 4.3) and are also syntactic synthesized.

To solve the unknown integers `l` and `u`, Synth mechanically generates SMT implications based on the type indexes. Since the positive index states that all elements `x` on the domain should satisfy `query x` and the negative that all elements outside of the domain should satisfy `True`, the following SMT constraint is mechanically generated:

```
∀ x. (x∈ dom ⇒  query x) ∧ (x∉ dom ⇒ True)
```

Solving such constraints gives us a value for `dom` if a solution exists. In practice, however, such solutions are often just a point, *i.e.,* the abstract domain contains only one secret. Although this is a correct solution, it is not precise. To increase precision we add optimization directives to constraints depending on the type of our approximation. That is, for $i \in \{1 \ldots n\}$ we add `maximize` u_i - l_i or `minimize` u_i - l_i for under-approximations and over-approximations respectively. These optimization constraints are handed to an SMT solver that supports optimization directives [5] and the produced model is an intended solution for `dom`. We used the Pareto optimizer of Z3 [5], such that no single optimization objective dominates the solution.

5.3 IterSynth: Iterative Synthesis of PowerSets

Powerset abstract domains (§ 4.4) are synthesized by Algorithm 1 that iteratively increments the powersets with individual intervals to avoid scalability problems faced by Z3 when optimizing multiple intervals at once.

Algorithm 1 Iterative Synthesis of Powersets

```
procedure IterSynth(k, $n$, $\tau$, apx)
  dom_i ← [Synth ($\mathcal{A}_\mathbb{P}$ [□] [] _ _)::$\tau$ n apx]
  dom_o ← []
  for i = 2 to k do
    if apx == under then
      dom_t ← Synth ($\mathcal{A}_\mathbb{P}$ (dom_i ++ □) dom_o _ _)::$\tau$ n apx
      dom_i ← dom_i ++ [dom_t]
    else
      dom_t ← Synth ($\mathcal{A}_\mathbb{P}$ dom_i (dom_o ++ □) _ _)::$\tau$ n apx
      dom_o ← dom_o ++ [dom_t]
    end if
  end for
  return ($\mathcal{A}_\mathbb{P}$ dom_i dom_o _ _)
end procedure
```

The algorithm takes as arguments the number of intervals `k` to be included in the powerset, the number of fields in the secret n, the refinement type of the powerset domain τ, and the kind of approximation `apx` (`under` or `over`). It first runs Synth (§ 5.2) to generate the first interval, with the top level type properly propagated to the hole. If this is for an under-approximation, more such intervals can be added to the powerset to boost the precision. Conversely, if the first synthesized interval is an over-approximation, then more intervals can be eliminated from the powerset to return a more precise over-approximation. At each iteration, the algorithm creates a new placeholder interval □, and Synth solves it, incrementally building up the inclusion list `dom_i`, or the exclusion list `dom_o`. Finally, the powerset is returned after k iterations. This is Anosy general synthesis algorithm since for $k = 1$ the returned powerset has a single interval.

As a final step, the returned powerset is lifted to the Haskell source and substituted in the sketch in § 5.1, which as a sanity check is validated by Liquid Haskell.

Discussion. Traditional abstract interpretation based techniques will refine the domains, as the query is evaluated with small step semantics, leading to imprecision at each step. In contrast, Anosy is more precise (as we show in § 6), because the final abstract domain is synthesized in the final step after accumulating constraints. However, Z3 does not give precise solutions when there are too many maximize/minimize directives (more than 6 in our experience), and does not handle non-linear objectives well. We leave exploration of better optimization algorithms to future work.

6 Evaluation

We empirically evaluated Anosy's performance using two case studies. In the first one (§ 6.1), we analyze efficiency and precision of Anosy when verifying and synthesizing ind. sets using a set of micro-benchmarks from prior work. In the

Table 1. Number of fields in the secret, and size of the precise ind. sets x/y for our benchmarks, where x and y denotes the number of secrets that evaluate to `True`, and `False` respectively.

#	Name	No. of fields	Size of ind. sets
B1	Birthday	2	259 / 13246
B2	Ship	3	1.01e+06 / 2.43e+07
B3	Photo	3	4 / 884
B4	Pizza	4	1.37e+10 / 2.81e+13
B5	Travel	4	2160 / 6.72e+06

second one (§ 6.2), we use the Anosy monad to construct an application that performs multiple queries (similar to those of § 2) while enforcing a security policy on the attacker's knowledge. With this case study, we evaluate how losses of precision introduced by Anosy's abstract domains affect the ability of answering multiple queries.

Experimental setup. Anosy is a GHC plugin built against GHC 8.10.1. All refinement types were verified with LiquidHaskell 0.8.10. Z3 4.8.10 was used to synthesize the bounds of the abstract domains. All experiments were performed on a Macbook Pro 2017 with 2.3 GHz Intel Core i5 and 8GB RAM.

6.1 Verification & Synthesis of ind. sets

In this case study, we analyze the Anosy's performance with respect to the verification and synthesis of ind. sets.

Benchmark Programs. Our benchmarks consist of 5 problems from Mardziel et al. [24], which represent a diverse set of queries (B3 and B4 come from a targeted advertisement case study from Facebook [9]):

(B1) *Birthday* checks if a user's birthday, the secret, is within the next 7 days of a fixed day[2].

(B2) *Ship* calculates if a ship can aid an island based on the island's location and the ship's onboard capacity.

(B3) *Photo* checks if a user would be possibly interested in a wedding photography service by checking if they are female, engaged, and in a certain age range.

(B4) *Pizza* checks if a user might be interested in ads of a local pizza parlor, based on their birth year, the level of school attended, their address latitude and longitude (scaled by 10^6).

(B5) *Travel* tests for a user interest in travels by checking if the user speaks English, has completed a high level of education, lives in one of several countries, and is older than 21.

For each problem, we encode the query as a Haskell function with the appropriate refinement type [44] where the secret domain is represented as a Haskell datatype for which we use the same bounds as [24]. Table 1 reports the number of fields in the secret, and the size of the precise ind. sets for each benchmark as x/y, where x denotes the size of the precise ind. set for the `True` response from query, and y is the size when the query responds `False`.

Experiment. For each benchmark, we use Anosy to (1) synthesize the under- and over-approximated ind. sets for both results `True` and `False` and (2) verify that the synthesized approximations match the refinement types from § 4. We run each benchmark 11 times to collect synthesis and verification times. We used a 10 sec timeout for each Z3 call. The goal is to evaluate the precision of the synthesized ind. set, and time taken for synthesis and verification to run.

Intervals. Figure 5a reports the results of our experiments for both the under- and over-approximated ind. sets using the interval abstract domain. Specifically, column *Size* reports the number of secrets in the approximated ind. set, column *Verif. time* reports the time (in sec) LiquidHaskell takes to verify the posteriors, and column *Synth. time* reports the time (in sec) taken for synthesizing the approximate ind. sets. The *% diff.* column lists the difference in size of the approximate ind. sets with the exact ones from Table 1. The lower the *% diff.* column value, the more precise is Anosy's synthesized ind. set, *i.e.,* it is closer to the ground truth.

For all our benchmarks, LiquidHaskell quickly verifies the correctness of the posteriors, in less than 4 sec on average. In some cases, like B1 and B3, Anosy can synthesize the exact ind. set for the `True` result using a single interval (for both approximations). For the `False` set, however, the tool returns an approximated result because the precise ind. set is not representable using intervals.

In 7 out of 10 synthesis problems, Anosy synthesizes the approximations in less than 5 seconds. The three outliers are the synthesis of under-approximations for the B2 and the synthesis of both approximations for the B4. B2 uses a relational query that creates a dependency between two secret fields, where the multi-objective maximization employed by Z3 runs longer. B4 uses very large bounds (in the orders of 10^8) which result in Z3 quickly finding a sub-optimal model but timing-out before finding an optimal solution.

Powersets of intervals. Figure 5b reports the results of our experiments using the powersets domain with 3 intervals. A higher number gives more precision for representation of the ind. set at the cost of taking more time for synthesis, due to our iterative synthesis algorithm (§ 5.3).

For under-approximations, Anosy successfully synthesizes both exact ind. sets for B1 using powersets, even though the `False` set was not representable using just a single interval. For B2 and B3, the powersets significantly improve precision, *i.e.,* we synthesize larger under-approximations.

This can be seen by comparing the *% diff.* column between Figure 5a and 5b, where the latter reports lower percentage differences from ground truth. In fact, for B3, Anosy can almost synthesize the entire ind. set for `False` with powersets of size 3, and it can synthesize the exact ind. set with powersets of size 4 (not shown in Figure 5b). For B4, powersets only marginally improve precision due to SMT optimization timing out. For over-approximations, we observe a similar

[2] We only use the deterministic version of the *Birthday* problem.

#	Under-approximation				Over-approximation			
	Size	% diff.	Verif. time	Synth. time	Size	% diff.	Verif. time	Synth. time
B1	259 / 9620	0 / 27	2.78 ± 0.03	1.11 ± 0.01	259 / 13505	0 / 2	2.64 ± 0.03	1.07 ± 0.01
B2	2.21e+05 / 1.01e+07	78 / 58	3.62 ± 0.02	9.26 ± 0.04	2.02e+06 / 2.54e+07	100 / 5	3.17 ± 0.02	4.00 ± 0.12
B3	4 / 664	0 / 25	3.12 ± 0.06	0.90 ± 0.07	4 / 888	0 / 0	2.83 ± 0.03	0.90 ± 0.01
B4	3.53e+04 / 1.35e+05	100 / 100	3.66 ± 0.04	20.92 ± 0.11	9.22e+12 / 2.81e+13	67200 / 0	3.29 ± 0.08	10.87 ± 0.01
B5	360 / 5.04e+06	83 / 25	3.81 ± 0.04	1.38 ± 0.04	35460 / 6.72e+06	1542 / 0	3.47 ± 0.04	0.89 ± 0.01

(a) Interval abstract domain

#	Under-approximation				Over-approximation			
	Size	% diff.	Verif. time	Synth. time	Size	% diff.	Verif. time	Synth. time
B1	259 / 13246	0 / 0	4.51 ± 0.05	1.13 ± 0.02	259 / 13505	0 / 2	4.34 ± 0.03	1.08 ± 0.01
B2	6.78e+05 / 1.62e+07	33 / 33	5.32 ± 0.09	14.34 ± 0.11	1.80e+06 / 2.54e+07	78 / 5	5.17 ± 0.02	4.89 ± 0.09
B3	4 / 880	0 / 0	5.29 ± 0.09	1.07 ± 0.03	4 / 888	0 / 0	4.99 ± 0.03	1.03 ± 0.01
B4	3.88e+05 / 4.00e+05	100 / 100	5.78 ± 0.03	54.89 ± 0.23	9.22e+12 / 2.81e+13	67200 / 0	5.48 ± 0.08	30.57 ± 0.07
B5	720 / 6.70e+06	67 / 0	6.02 ± 0.07	13.26 ± 0.09	6300 / 6.72e+06	192 / 0	5.96 ± 0.04	15.25 ± 0.03

(b) Powerset of intervals with size 3

Figure 5. Ind. sets synthesis and verification of posteriors. Column *Size* reports the size of the synthesized ind. sets, where x is the size of the `True` set and y of the `False` set in x/y. *% diff* shows the percentage difference of the size from precise ind. set in Table 1 (lower value is better). *Verif. time* and *Synth. time* columns report (in seconds) the median and the semi-interquartile over 11 runs.

increase in precision, in particular in B3 and B5 where the synthesized approximations are close to the exact values. B4 slows down drastically because synthesis of each interval takes almost 10 seconds due to SMT timeouts.

Discussion. We note that the time taken to synthesize the ind. set is higher than running a static analysis tool by giving a prior such as `Prob` [24]. On our benchmarks we noticed that synthesis takes about 54.2x longer, than running `Prob`. However, for Anosy this is an one-time cost: once the ind. set is synthesized for a query—it can be used by the application with any number of priors—without running any expensive static analysis. Moreover, in contrast to `Prob` , Anosy can automatically split regions into intervals such that their union in the powerset gives a better accuracy. This shows up in the Figure 5b where just size 3 is enough to get us the exact ind. set for a few benchmark (*% diff.* is 0). In our experience, SMT-based synthesis works well for queries that contain point-wise comparisons (*e.g.*, query checking if the secret is one of a few constants).

6.2 Secure Advertising System

In this case study, we go back to the advertisement example in § 2 which we implement using Anosy to restrict the information leaked through `downgrade` . The goal of this case study is evaluating how the choice of abstract domains affects the number of declassification queries authorized by Anosy.

Application. We implemented the advertising query system from § 2 in Haskell using the `AnosyT` monad, with the `UserLoc` type as the secret. The system executes a sequence of 50 queries (one per restaurant branch): we use the `nearby` query from § 2 with the origin, denoting in this experiment the location of the restaurant, being a randomly generated point in the 400 × 400 space.

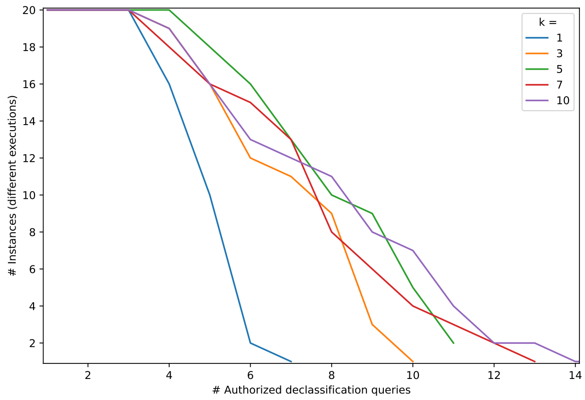


Figure 6. The lines show the number of execution instances (Y-axis) that were authorized for the i-th declassification query (X-axis). Each line corresponds to the under-approximated ind. set of powersets of size k.

Security policy and enforcement. Our program implements the security policy `qpolicy` from § 2, which restricts the restaurant chain from learning the user location below a set of 100 possible locations. To easily enforce the security policy, we wrapped the advertising query in the `downgrade` operation of `AnosyT` as in § 3.

Initially, our system starts with a prior knowledge equivalent to the entire secret domain 400 × 400 (i.e., the attacker does not have any information about the secret). As the system executes queries, the AnosyT monad tracks an under-approximation of the attacker's posterior knowledge based on the query result and on the prior. If the posterior complies with the policy, then the monad outputs the query result and the system continues with the next query. If a policy violation is detected, the system terminates the execution.

Experiment. For each experiment, we generate a new user location randomly, used as the secret, in the 400 × 400 space, and we run through the 50 queries for every restaurant location. For each execution, we measure after how many queries the system stops due to a policy violation. We repeat this experiment 20 times, to get the mean and standard deviation of query count and discuss them below.

Results. Figure 6 reports the results of our experiments. The line for each k, *i.e.,* the number of synthesized intervals in the powerset, depicts the number of experiment instances that are still running (Y-axis) after executing the i-th query (X-axis). For example, in the $k = 1$ powerset (equivalent to an interval), the system was able to answer the first 3 queries in all 20 instances without violating the policy, but only 2 instances were able to answer the 6th query.

As the size of powersets increases (from 3 to 10), the system can compute more precise under-approximations and, therefore, securely answer more queries, as can be seen in the figure. Specifically, for powerset of size $k = 3$, the system answers a maximum of 10 queries over 20 runs, with only 1 run reaching the 10th query. Similarly, the maximum number of queries answered increases to 14, due to increased precision by using powersets of size 10. Moreover, more than 10 instances answer more than 6 queries if the size of powersets goes above 3. This shows that Anosy can be used to build a practical system, that can answer multiple queries with precision without violating the declassification policy.

We note that the intersection of powersets made of k_1 and k_2 intervals produces a powerset of $k_1 k_2$ intervals, many of which are small or empty. This causes under-performance of higher sized powerset in intermediate steps, such as $k = 5$ performs better around the steps 5 to 7 than higher ks. However, over longer sequence of queries a higher sized powerset performs better.

7 Related Work

Information-flow control. Language-based information-flow control (IFC) [33] provides principled foundations for reasoning about program security. Researchers have proposed many enforcement mechanisms for IFC like type systems [1, 6, 11, 21, 29, 31, 32], static analyses [16], and runtime monitors [13] to verify and enforce security properties like non-interference. The ind. sets and knowledge approximations computed by Anosy can be used as a building block to enforce both non-interference as well as more complex security policies, as we discuss below.

Use of knowledge in IFC. The notion of attacker knowledge has been originally introduced to reason about dynamic IFC policies, where the notion of "public" and "secret" information can vary during the computation [2, 13, 40]. The notion of *belief* consists of a knowledge, *i.e.,* set of possible secret values, equipped with a probability distribution describing how likely each secret is. Existing approaches [14, 19, 24, 39] can enforce security policies involving probabilistic statements over an attacker's belief, *e.g.,* "an attacker cannot learn that a secret holds with probability higher than 0.7". We plan to deal with probability distributions in future work.

Quantitative Information Flow approaches provide quantitative metrics, *e.g.,* Shannon entropy [35], Bayes vulnerability [36], and guessing entropy [25], that summarize the amount of leaked information. For this, several approaches [3, 8, 18] first compute a representation of a program's indistinguishability equivalence relation, whereas we represent the partition induced by the indistinguishability relation, where each ind. set is one of the relation's equivalence classes.

There are several approaches for approximating the indistinguishability relation in the literature. Clark et al. [8] provide techniques to approximate the indistinguishability relation for straight line programs. Backes et al. [3] automates the synthesis of such equivalence relations using program verification techniques, and Köpf and Rybalchenko [18] further improve the approach by combining it with sampling-based techniques. Similarly to [3], we automatically synthesize ind. sets from programs. In contrast to [3, 8, 18], the correctness of our ind. sets is also automatically and machine-checked.

Declassification. Declassification is used in IFC systems to selectively allow leaks, and several extensions of non-interference account for it [2, 13, 40]; we refer the reader to [34] for a survey of declassification in IFC. Most systems treat declassification statements as *trusted*. Our work focuses on the *what* dimension of declassification, that is, our policies restrict *what* information can be declassified. In contrast, Chong and Myers [7] enforce declassification policies that target other aspects of declassification, specifically, limiting in which context declassification is allowed and how data can be handled after declassification.

Program Synthesis. Anosy's synthesis technique follows sketch-based synthesis [37], where traditionally users provide a partial implementation with *holes* and some specifications based on which the synthesizer fills in the holes. Standard types have extensively served as a synthesis template often combined with tests, examples, or user-interaction [12, 15, 23, 26]. Refinement types provide stronger specifications, thus, as demonstrated by Synquid [28], do not require further tests or user information. In Anosy, we use the refinement type synthesis idea of Synquid, but also mechanically generate the knowledge specific refinement types.

References

[1] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. 2012. Sharing mobile code securely with information flow control. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[2] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 207–221.

[3] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic Discovery and Quantification of Information Leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 141–153.

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2006. Widening operators for powerset domains. In *J Softw Tools Technol Transfer*. Springer.

[5] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *vz*-an optimizing SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 194–199.

[6] Niklas Broberg, Bart van Delft, and David Sands. 2017. Paragon–Practical programming with information flow control. *Journal of Computer Security* 25, 4-5 (2017), 323–365.

[7] Stephen Chong and Andrew C Myers. 2004. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*. 198–209.

[8] David Clark, Sebastian Hunt, and Pasquale Malacaria. 2005. Quantitative Information Flow, Relations and Polymorphic Types. *J. Log. Comput.* 15, 2 (2005), 181–199.

[9] Adele Cooper. 2011. Facebook Ads: A Guide to Targeting and Reporting. https://web.archive.org/web/20110521050104/http://www.openforum.com/articles/facebook-ads-a-guide-to-targeting-and-reporting-adele-cooper.

[10] Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod.

[11] D. Devriese and F. Piessens. 2011. Information flow enforcement in monadic libraries. In *TLDI*. https://doi.org/10.1145/1929553.1929564

[12] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Vol. 52. ACM New York, NY, USA, 422–436.

[13] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. 2019. Information-flow Control for Database-backed Applications. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy*. IEEE.

[14] Marco Guarnieri, Srdjan Marinovic, and David Basin. 2017. Securing Databases from Probabilistic Inference. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium*. IEEE, 343–359.

[15] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and Effect-Guided Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3453483.3454048

[16] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM.

[17] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

[18] Boris Köpf and Andrey Rybalchenko. 2010. Approximation and Randomization for Quantitative Information-Flow Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society, 3–14.

[19] Martin Kucera, Petar Tsankov, Timon Gehr, Marco Guarnieri, and Martin Vechev. 2017. Synthesis of Probabilistic Privacy Enforcement. In *Proceedings of the 24th ACM Conference on Computer and Communications Security*. ACM, 391–408.

[20] N. Lehmann, R. Kunkel, J. Brown, J. Yang, D. Stefan, N. Polikarpova, R. Jhala, and N. Vazou. 2021. STORM: Refinement Types for Secure Web Applications. (2021). To appear in USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[21] P. Li and S. Zdancewic. 2006. Encoding information flow in Haskell. In *CSFW*. https://doi.org/10.1109/CSFW.2006.13

[22] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

[23] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[24] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. 2013. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *J. Comput. Secur.* 21, 4 (2013), 463–532. https://doi.org/10.3233/JCS-130469

[25] James L Massey. 1994. Guessing and entropy. In *Proceedings of 1994 IEEE International Symposium on Information Theory*. IEEE, 204.

[26] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* 50, 6 (2015), 619–630.

[27] James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information flow security for multi-tier web applications. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[28] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* 51, 6, 522–538.

[29] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid information flow control. *Proc. ACM Program. Lang.* 4, ICFP (2020), 105:1–105:30.

[30] Corneliu Popeea and Wei-Ngan Chin. 2006. Inferring Disjunctive Postconditions. In *Advances in Computer Science (ASIAN '06)*. Springer.

[31] François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 319–330.

[32] Alejandro Russo. 2015. Functional pearl: two can keep a secret, if one of them uses Haskell. *ACM SIGPLAN Notices* 50, 9 (2015), 280–288.

[33] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19.

[34] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *Journal of Computer Security* 17, 5 (2009), 517–548.

[35] Claude Elwood Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review* 5, 1 (2001), 3–55.

[36] Geoffrey Smith. 2009. On the foundations of quantitative information flow. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 288–302.

[37] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs.

In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[38] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*. 95–106.

[39] Ian Sweet, José Manuel Calderón Trilla, Chad Scherrer, Michael Hicks, and Stephen Magill. 2018. What's the Over/Under? Probabilistic Bounds on Information Leakage. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10804)*, Lujo Bauer and Ralf Küsters (Eds.). Springer, 3–27.

[40] Bart van Delft, Sebastian Hunt, and David Sands. 2015. Very static enforcement of dynamic policies. In *International Conference on Principles of Security and Trust*. Springer, 32–52.

[41] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). *SIGPLAN Not.* 53, 7 (sep 2018), 132–144. https://doi.org/10.1145/3299711.3242756

[42] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228.

[43] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. (2014).

[44] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–31.