Anosy: Approximated Knowledge Synthesis with Refinement Types for Declassification

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 power sets 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 power sets 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...
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.

```haskell
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:

```haskell
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 \((x_{org}, y_{org})\).

```haskell
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.

```haskell
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 downgrades, 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,
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.

\[
\text{qpolicy dom = size dom > 100}
\]

This policy will allow decclassifying 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 decclassification 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 decclassification knowledge. To do 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).

\[
\text{secret } \leftarrow \text{getUserLoc secret (nearby (200,200))}
\]
\[
\text{kn1 } \leftarrow \text{downgradeUs secret (nearby (200,200))}
\]
\[
\text{kn2 } \leftarrow \text{downgradeUs secret (nearby (400,200))}
\]
\[
\text{-- if kn1 } \land \text{ kn2 then secret } = (300,200)
\]

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.

\[
\text{qpolicy dom = size dom > 100}
\]

This policy will allow decclassifying 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 decclassification 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 decclassification knowledge. To do 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).

\[
\text{secret } \leftarrow \text{getUserLoc secret (nearby (200,200))}
\]
\[
\text{kn1 } \leftarrow \text{downgradeUs secret (nearby (200,200))}
\]
\[
\text{kn2 } \leftarrow \text{downgradeUs secret (nearby (400,200))}
\]
\[
\text{-- if kn1 } \land \text{ kn2 then secret } = (300,200)
\]

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:

\[
\text{data AInt } = \text{AInt [lower :: Int, upper :: Int]}
\]
\[
\text{data A } = \text{A [AInt]}
\]
\[
\text{under_indset } :: (A, A)
\]
\[
\text{under_indset } = (A [AInt 121 279, AInt 179 221],
\]
\[
\text{ A [AInt 0 400, AInt 0 99])}
\]

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 \( \cap \) 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.

\[
\text{underapprox} :: \mathcal{A} \rightarrow (\mathcal{A}, \mathcal{A})
\]

\[
\text{underapprox} \ p = (p \cap \text{trueInd}, p \cap \text{falseInd})
\]

where \((\text{trueInd}, \text{falseInd}) = \underindset\)

The intersection \( \cap \) 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, \(\mathcal{A} <\{\lambda \rightarrow \theta \land 1\} \) denotes the abstract domain whose elements are positive values. Using this abstraction, we specify the ind. set and knowledge approximations as follows:

\[
\text{underindset} :: (\mathcal{A} <\{\lambda \rightarrow \text{query 1}\}, \mathcal{A} <\{\lambda \rightarrow \neg \text{query 1}\})
\]

\[
\text{underapprox} :: p: \mathcal{A} \rightarrow (\mathcal{A} <\{x \rightarrow \text{query x} \land (x \in p)\}, \mathcal{A} <\{x \rightarrow \neg \text{query x} \land (x \in p)\})
\]

The \( \underindset \) 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 \( \text{underapprox} \) 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 \( \text{underapprox} \) 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 \( \text{UserLoc} \) contains two integer fields, the template of \( \underindset \) is the following program sketch [37]:

\[
\underindset = (\mathcal{A} [\mathcal{A} \int l_1 u_1, \mathcal{A} \int l_2 u_2], \mathcal{A} [\mathcal{A} \int l_1 u_1, \mathcal{A} \int l_2 u_2])
\]

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:

\[
\forall x, y. l_1 \leq x \leq u_1 \land l_2 \leq y \leq u_2 \implies \text{query}(x, y) \quad \text{(Under-approx, True)}
\]

\[
\forall x, y. l_1 \leq x \leq u_1 \land l_2 \leq y \leq u_2 \implies \neg\text{query}(x, y) \quad \text{(Under-approx, False)}
\]

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 increase 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 kno-
wledge. 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 = [0, 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 underly-
ing 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 associ-
ing 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 policy (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 downgrad-
ing 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 trans-
former [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 T 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 knowledge1. Then, it

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

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 T
        $ 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.

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.

Correctness: Policy Enforcement. Suppose a secret s that
has been downgraded n times by the queries query1, . . . , queryn.
After each downgrade, the knowledge is refined. So,
starting from the top knowledge (K0 ⊆ ⊤), after n queries,
the knowledge evolves as follows: K0 ⊆ K1 ⊆ · · · ⊆ Kn ⊆
. . . ⊆ Kn, where Kn = Kn−1 ∩ \{x | queryi, x = queryi, s\}. 496

497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550
We can show that for each \(i\)-th downgrade of the secret \(s\), there exists a posterior \(P_i\) so that \((s, P_i)\) is in the secrets map and also \(P_i\) is an under-approximation of the knowledge \(K_i\), that is \(P_i \subseteq K_i\). The proof goes by induction on \(i\), assuming that the attacker and the downgrade implementation start from the same \(\top\) knowledge, and the inductive step relies on the specification of the \(\text{approx}\) function and the way downgrades modify secrets, \(i.e\), using \(\text{post}\top\) or \(\text{post}\bot\) depending on the response of the query.

Thus if our quantitative policy enforces a lower bound on the size of the leaked knowledge, \(e.g\), \(\text{qpolicy dom = size dom} > k\) it is correctly enforced by \(\text{downgrade}\): since \(\mathcal{P}_i \subseteq \mathcal{K}_i\), then \(\text{qpolicy} \mathcal{P}_i\) implies \(\text{qpolicy} \mathcal{K}_i\) at each stage of the execution. Note that for correctness of policy implementation, 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 \(\text{post}\top\) and \(\text{post}\bot\), 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 \(\text{approx}\) that correctly computes the underapproximation of posterior knowledge. Here, we show how refinement types can specify correctness of \(\text{approx}\), 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 \(\text{AbstractDomain a s where}\) abstract refined type class interface stating that \(a\) can abstract, \(i.e\), represent a set of values of \(s\). For example, an instance \(\text{instance } \mathcal{A}_1 \text{ UserLoc}\) states that the data type \(\mathcal{A}_1\) (that we will define in § 4.3) abstracts \(\text{UserLoc}\) (of § 2.1). The interface contains method definitions and class laws, and when required the abstract domain is indexed by abstract refinements.

\textbf{Class Methods}. The class contains six, standard, set-theoretic methods. Top (\(\top\)) and bottom (\(\bot\)), 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, \(\text{size } s\) computes the number of concrete values represented by an abstract domain.

\begin{align*}
\text{class AbstractDomain a s where} & \\
\top & : a < (\lambda . \text{True}, \lambda . \text{False}) > \\
\bot & : a < (\lambda . \text{False}, \lambda . \text{True}) > \\
s & : s \rightarrow a \rightarrow \text{Bool} \\
\subseteq & : a \rightarrow a \rightarrow \text{Bool} \\
\cap & : d_1, d_2 : a < p_1, n_1 > \rightarrow d_2 : a < p_1, n_1 > \\
& \quad \quad \rightarrow (d_1 : a < p_1, n_1 \cup n_2 > | d_1 \subseteq d_2 \land d_2 \subseteq d_3) \\
\text{size} & : a \rightarrow \{i : \text{Int} | 0 \leq i\} \\
\text{-- class laws} & \\
\text{sizeLaw} & : d_1, a \rightarrow (d_2, a | d_1 \subseteq d_2) \\
& \quad \quad \rightarrow \{\text{size } d_1 \leq \text{size } d_2\} \\
\text{subsetLaw} & : c : s \rightarrow d_1, a \rightarrow (d_2, a | d_1 \subseteq d_2) \\
& \quad \quad \rightarrow \{c \in d_1 \Rightarrow c \in d_2\} \\
\end{align*}

\textbf{Figure 3. Abstraction Domain Type Class}

\textbf{Class Laws}. We use refinement types to specify two class laws that should be satisfied by the \(\subseteq\) and \(\text{size}\) methods. \(\text{sizeLaw}\) 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 \(d_2\). \(\text{subsetLaw}\) 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.

\textbf{Abstract Indexes}. In the types of top, bottom, and intersection, the type \(a\) is indexed by two predicates \(p\) and \(n\) \((s \rightarrow \text{Bool})\). 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 \prec p, n > \{(d : a | \forall x. x \in d \Rightarrow p x \land \forall x. x \notin d \Rightarrow 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 \(\top\) states that the positive predicate is \(\text{True}\), \(i.e\), satisfied by all elements of the domain, and the negative \(\text{False}\), \(i.e\), no elements are outside of the domain. Similarly, the empty domain \(\bot\) has a \(\text{False}\) positive predicate, \(i.e\), no elements are in the domain, and \(\text{True}\) 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 \(d_2\) \(i.e\), \(p_1 \cup p_2\). The negative predicate indicates points excluded from \(d_3\) are points excluded from either \(d_1\) or \(d_2\), \(i.e\), \(n_1 \cap n_2\). The refinement on \(d_3\) ensure that \(d_3\) is a subset \(\subseteq\) of both \(d_1\) and \(d_2\). For abstract
The positive predicates are just true. The domains can include any number of secrets as long as they are not leaving out.

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 over-approximations.)

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 M, the interval abstract domain that can abstract any secret type 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). M is defined as follows:

```
→ S = Int × Int × ...
data MInt = MInt (lower :: Int, upper :: Int)
type Proof p x = (v :: S → P | v = x )
data M <p :: S → Bool, n :: S → Bool>
  = M (dom :: [MInt], pos :: x :: S | x ∈ dom ) → Proof p x
  | ⊥ (neg :: x :: S → Proof n x )
```

M has three constructors. T and ⊥ respectively denote the complete and empty domains. M represents the domain of any n-dimensional intervals, where n is the length of dom. An interval MInt represents integers between lower and upper. For a secret s = s1 ∗ s2 ∗ ... sn, an M represents each s by the i th element of its dom (si ∈ (dom!i)) in the n dimensional space. For example, domEx = [([MInt 188 212],
(MInt 112 288)]) is the rectangle of x ∈ [188,212], y ∈ [112,288] in the two dimensional space of UserLoc.

Proof Terms. The pos and neg components in the M definition are proof terms that give meaning to the positive p and negative n abstract refinements. The complete domain T contains the proof field pos that states that every secret s
should satisfy the positive predicate \( p \) (i.e., \( x : S \rightarrow \text{Proof} \\neg p \ x \)), and the empty domain contains only the proof \( \text{neg} \) 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 here 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 \( \text{don} \) should satisfy \( p \), while the neg field states that each \( x \) that does not belong to \( \text{dom} \) 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 \( \text{domEx} \) (described above) only represents elements that are nearby \( (200,200) \).

```haskell
example :: \( \mathcal{A}_I \) <\( \{ s \rightarrow \text{nearby} (200,200) \ s, \text{true} \} \>
example = \( \mathcal{A}_I \) \( \text{domEx} \) \( \text{exPos} \) \( \\backslash x \rightarrow x \)

exPos :: \( s : \{ \text{UserLoc} \mid s \in \text{domEx} \} \rightarrow \{ o : \text{UserLoc} \mid \text{nearby} (200,200) \ s \wedge o = s \} \)

\( \text{exPos} \) (\( \text{UserLoc} \ x \ y \)) = \( \text{UserLoc} \ x \ y \)
```

The proof term \( \text{exPos} \) is an identity function refined to satisfy the pos specification. Once the type signature of \( \text{exPos} \) 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. \( \epsilon \) checks if any secret is between lower and upper for every dimension. \( \subseteq \) checks if the intervals representing the first argument is included in the intervals representing the second argument. \( \cap \) 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 \( \mathcal{A}_P \) i.e., a set of interval domains. Similar to intervals, powerset \( \mathcal{A}_P \) is also parameterized with the positive and negative predicates:

```haskell
data \( \mathcal{A}_P \) <\( p : S \rightarrow \text{Bool} \), \( n : S \rightarrow \text{Bool} \) > = \( \mathcal{A}_P \) (
  \( \text{dom}_i : [\mathcal{A}_I] \), \( \text{dom}_o : [\mathcal{A}_I] \),
  \( \text{pos} : x : \{ S | x \in \text{dom}_i \wedge x \notin \text{dom}_o \} \rightarrow \text{Proof} \ p \ x \),
  \( \text{neg} : x : \{ S | x \notin \text{dom}_i \wedge x \in \text{dom}_o \} \rightarrow \text{Proof} \ n \ x \)
)
```

\( \mathcal{A}_P \) contains four fields. \( \text{dom}_i \) is the set (represented as a list) of intervals that are contained in the powerset. \( \text{dom}_o \) 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 \( T \) and bottom \( \bot \) for \( \mathcal{A}_P \) as they can be represented using \( T_I \) or \( L_I \) 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 (\( \epsilon \)) the powerset \( \mathcal{A}_P \) if it belongs to any individual interval of the \( \text{dom}_i \) list but not in any individual interval of the \( \text{dom}_o \) list. The subset \( d_1 \subseteq d_2 \) operation checks if all the individual intervals in the inclusion list \( \text{dom}_i \) of \( d_1 \) is a subset of at least one interval in the inclusion list \( \text{dom}_i \) of \( d_2 \), and also that none of the individual intervals in the exclusion list \( \text{dom}_o \) of \( d_1 \) is a subset of any interval in \( \text{dom}_o \) of \( d_2 \). This operation returns \( \text{True} \) is the first powerset is a subset of the second, but if it returns \( \text{False} \) 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 \( \text{dom}_i \) of \( d_1 \) and \( \text{dom}_i \) of \( d_2 \), and the exclusion interval list is simply the union of all intervals in the individual exclusion lists \( \text{dom}_o \) of \( d_1 \) and \( \text{dom}_o \) 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 individual 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:

\[
\underindset = (\varnothing :: A \leftarrow (x \rightarrow \text{query } x, \varnothing \rightarrow \text{True})), \quad \varnothing :: A \leftarrow (x \rightarrow \text{query } x, \varnothing \rightarrow \text{True})
\]

Following the structure of the type we simply introduce typed holes of the form \( \varnothing :: t \) 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 underindset as an interval domain.

\[
\varnothing :: A_i \leftarrow (x \rightarrow \text{query } x, \varnothing \rightarrow \text{True}), \quad \varnothing = A_i \text{ dom pos neg}
\]

The solution is using the \( A_i \) applied to the domain list \( \text{dom} \) and the pos and neg proof terms. The \( \text{dom} \) is a list of \( A\text{Int} \) 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 \( l_i \) and \( u_i \), SYNTH mechanically generates SMT implications based on the type indexes. Since the positive index states that all elements \( x \) on the domain should satisfy \( \text{query } x \) and the negative that all elements outside of the domain should satisfy \( \text{True} \), the following SMT constraint is mechanically generated:

\[
\forall x. (x \in \text{dom} \Rightarrow \text{query } x) \land (x \notin \text{dom} \Rightarrow \text{True})
\]

Solving such constraints gives us a value for \( \text{dom} \) 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 \( \text{dom} \). 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.

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 \( r \), and the kind of approximation \( \text{apx} \) (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 \( \varnothing \), and SYNTH solves it, incrementally building up the inclusion list \( \text{dom}_i \), or the exclusion list \( \text{dom}_o \). 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>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>No. of fields</th>
<th>Size of ind. sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Birthday</td>
<td>2</td>
<td>259 / 13246</td>
</tr>
<tr>
<td>B2</td>
<td>Ship</td>
<td>3</td>
<td>1.01e+06 / 2.43e+07</td>
</tr>
<tr>
<td>B3</td>
<td>Photo</td>
<td>3</td>
<td>4 / 884</td>
</tr>
<tr>
<td>B4</td>
<td>Pizza</td>
<td>4</td>
<td>1.37e+10 / 2.81e+13</td>
</tr>
<tr>
<td>B5</td>
<td>Travel</td>
<td>4</td>
<td>2160 / 6.72e+06</td>
</tr>
</tbody>
</table>

In this case study, we analyze the Anosy’s performance with respect to the verification and synthesis of ind. sets. 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 $10^6$) 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 powerset 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

---

*2We only use the deterministic version of the Birthday problem.*
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 `downgrad`. 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 $400 \times 400$ space.

**Security policy and enforcement.** Our program implements the security policy `apolicy` 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 `downgrad` operation of AnosyT as in § 3.

Initially, our system starts with a prior knowledge equivalent to the entire secret domain $400 \times 400$ (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 x 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 k1 and k2 intervals produces a powerset of k1k2 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 SynQUD [28], do not require further tests or user information. In Anosy, we use the refinement type synthesis idea of SynQUD, but also mechanically generate the knowledge specific refinement types.
References


