Formalizing Algorithmic Bounds in the Query Model in EasyCrypt

Alley Stoughton
Boston University, MA, USA

Carol Chen
Stuyvesant High School, New York, NY, USA

Marco Gaboardi
Boston University, MA, USA

Weihao Qu
Boston University, MA, USA

Abstract

We use the EasyCrypt proof assistant to formalize the adversarial approach to proving lower bounds for computational problems in the query model. This is done using a lower bound game between an algorithm and adversary, in which the adversary answers the algorithm's queries in a way that makes the algorithm issue at least the desired number of queries. A complementary upper bound game is used for proving upper bounds of algorithms; here the adversary incrementally and adaptively realizes an algorithm's input. We prove a natural connection between the lower and upper bound games, and apply our framework to three computational problems, including searching in an ordered list and comparison-based sorting, giving evidence for the generality of our notion of algorithm and the usefulness of our framework.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Models of computation; Theory of computation → Design and analysis of algorithms

Keywords and phrases query model, lower bound, upper bound, adversary argument, EasyCrypt

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.30

Supplementary Material EasyCrypt Frameworks for Proving Algorithmic Bounds:
Software (Source Code): `https://github.com/alleystoughton/AlgorithmicBounds`

Funding *Alley Stoughton*: National Science Foundation Grant No. 1801564.
Marco Gaboardi: National Science Foundation Grant No. 2040249.

Acknowledgements We would like to thank Mark Bun for numerous helpful discussions. The anonymous referees provided very helpful feedback that helped us improve the paper.

1 Introduction

1.1 Algorithmic Background

The establishment of lower bounds for computational problems, and upper bounds for algorithms solving those problems, are synergistic activities. As algorithms and their upper bounds become better, that encourages the search for tighter lower bounds, and vice versa. One of the richest sources of non-trivial lower bounds is the query model,[1] in which an algorithm doesn't have direct access to its input, but may only issue queries about it. A lower bound theorem in the query model shows a lower bound, as a function of the input size, n, of the number of queries any algorithm must issue in the worst case, in order to solve a size-n instance of the given computational problem.

[1] We include comparison-based sorting in this category.

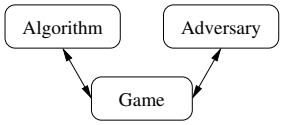

Figure 1 Game between Algorithm and Adversary.

Adversary arguments (the adversarial method) are one of the most common ways of establishing such lower bounds [3, 2, 13]. When Knuth first published Volume 3 of *The Art of Computer Programming* [15], he used the term "oracular lower bounds" for this method. In Section 5.3.2, when discussing minimum comparison merging, he talked of constructing a "suitable oracle" that answers an algorithm's queries about which of two elements is greater:

> If we can construct a suitable oracle ... we can ensure that every valid merging algorithm will have to ask a rather large number of questions.

In the 1998 Second Edition of Volume 3 [16], he has adopted the modern terminology of "adversary" instead of oracle.

An adversary argument is structured as a two-party game between the algorithm and adversary, as illustrated in Figure 1. The algorithm is allowed to query parts of the input (e.g., bits, or cells), and the adversary is programmed to answer in a way that delays the game long enough to achieve the desired lower bound. The duration of the game is measured by steps, where one step consists of a query by the algorithm combined with its answer by the adversary. The game[2] plays the role of referee, keeping track of the inputs that are consistent with the adversary's answers so far, and declaring the game to be over when all the remaining inputs have the same answer. Neither the algorithm nor the adversary have to report anything. When proving a lower bound theorem, one exhibits an adversary, and then proves that, for all algorithms (correct or not), the game continues for long enough to achieve the lower bound – in which case we say the adversary has "won" the game.

Perhaps the simplest example of an adversary argument is the proof that an algorithm computing the or (disjunction) function of a list of booleans of size n must query, in the worst case, the value of every index of the list, and so must make n queries. The adversary can be stateless, and when asked for the value of the ith element of the list it can always answer false, keeping the algorithm uncertain as to whether the result of the or function will be true or false. Initially the set of input lists maintained by the game consists of all 2^n lists of booleans of size n. As the game progresses, the set of input lists will always consist of all lists of booleans in which all the elements at the indices queried so far are false. Thus as long as not every index has been queried, there is at least one input list including at least one occurrence of true, and so where the answer is true, and exactly one input list where all the elements are false, and thus the answer is false. In other words, the game is not over until all indices have been queried.

In our work, we are interested in establishing concrete bounds (like n in the preceding example), rather than asymptotic ones. Of course concrete bounds can by abstracted to asymptotic bounds after the initial proof, as desired.

[2] We consider a slight variation of the framework of [2] that is well-suited to formalization.

1.2 Formalization in EasyCrypt

When we set out to formalize adversary arguments using a proof assistant, we opted to work in EasyCrypt because of our experience (e.g., [21]) formalizing cryptographic games in EasyCrypt. EasyCrypt (see Section 2) has a module system allowing one to implement algorithms and adversaries as modules – collections of procedures operating on private variables. This allows our *lower bound game* (see Section 3.3) to be expressed as a parameterized module – parameterized by an abstract algorithm and adversary. Procedures in EasyCrypt are allowed to employ random assignments, choosing values from probability distributions. Consequently, lower bound theorems in our formalization must be proved against potentially probabilistic algorithms. Making use of randomness doesn't give an algorithm an advantage, though, if we consider worst-case efficiency and expect it to produce the correct result with probability 1. An adversary "wins" a run of the lower bound game against an algorithm and for a bound lb iff the game runs at least lb steps before the queries/answers uniquely determine the computational problem's answer and the game ends. When proving a lower bound theorem, one exhibits an adversary and shows that, for all algorithms, the adversary wins the lower bound game with probability 1. The lower bound game itself isn't probabilistic, and neither are the adversaries we have used in our work to date. Thus when an algorithm is also non-probabilistic, winning with probability 1 just means the only possible run results in a win.

In our framework, we have a general method (see Section 3.1) for expressing computational problems in the query model over inputs consisting of lists of a fixed size, n. Our bounds are then expressions in terms of n. We accommodate restrictions on input lists, e.g., that they are sorted. Our notion of query is general enough to encompass comparisons, i.e., queries asking not for the *value* of the input at a given index, but asking how the values at two indices are *related*. A computational problem is parameterized by an auxiliary value picked by the adversary, but made available to the algorithm. E.g., this is used in searching problems to say which element should be searched for in an ordered list.

Upper bound theorems can also be naturally expressed in terms of a game: an *upper bound game*, parameterized by an algorithm and an adversary. This time (see Section 3.4), the algorithm reports an answer to the computational problem, in addition to issuing queries.[3] The adversary adaptively answers the algorithm's queries, incrementally realizing more and more of the input list (or the relative order of the input list's elements, in the case of comparison queries). This notion of adversary includes ones based on hard-coded auxiliary values and input lists. The upper bound game plays the role of referee, keeping track of the inputs that are consistent with the answers to the queries issued so far. An algorithm "wins" a run of the upper bound game for a bound ub and against an adversary iff, in no more than ub steps, either the algorithm reports the correct answer (the same, for all remaining consistent inputs), or the adversary answers a query inconsistently, causing the game to end early. When proving an upper bound theorem for an algorithm, we prove that for all (even probabilistic) adversaries, the algorithm wins the upper bound game against the adversary with probability 1. The upper bound game itself isn't probabilistic, and so when both the algorithm and adversary are also non-probabilistic, winning with probability 1 just means the only possible run results in a win.

We connect the lower and upper bound games via the following theorem (see Section 3.5 for the formal statement): If an adversary wins the lower bound game against an algorithm for bound lb with probability 1, and $ub < lb$, then with probability 1, the algorithm loses the upper bound game against the adversary for bound ub.

[3] Such an algorithm can be converted to the kind expected by the lower bound game.

We use examples to informally make the case that our model of algorithm – in which a game asks the algorithm for its next query, and later tells it the answer – is general enough to model all algorithms. In a recursive algorithm, e.g., the stack of recursive calls may be arbitrarily deep at the point where an answer is needed to a query. We must suspend the computation, storing the suspension in a global variable of the algorithm. Once the answer is received, this suspended computation can then be resumed and supplied the query's answer. We realize this using terms of an ad hoc functional language as the suspensions. We leave to future work the design of, and development of a meta-theory for, a general purpose functional language for expressing suspendable algorithms in the query model.

In our current work, the probabilistic nature of EasyCrypt is something of an impediment. Because we are modeling the worst-case query complexity of algorithms that must always produce correct results, randomization is not an advantage when expressing algorithms. Fortunately, we have a generic method for reducing lower and upper bound theorems for probability 1 to non-probabilistic Hoare logic judgments. This method still requires us to prove termination with probability 1 of the procedures of our adversary (resp., algorithm) in a lower bound (resp., upper bound) proof. But doing this is not significantly different from proving termination, as we don't use randomness in our adversaries (resp., algorithms). In future work, though, we would like to work with randomized algorithms that are allowed to produce incorrect results with small probability. Thus continuing to work in EasyCrypt may be an advantage.

By using our EasyCrypt framework for lower and upper bound results, we eliminate errors that could lurk in ad hoc formalizations. The lower and upper bound games act as referees, and EasyCrypt guarantees that algorithms and adversaries cannot interfere with the states of the games or their opponents. Assuming we are happy with the formalization of a computational problem, the definition of an adversary is simply part of the proof of a lower bound theorem; if it has "bugs", as long as the proof goes through, they are irrelevant. Furthermore, when proving an upper bound theorem for a given algorithm, the framework ensures that we are counting queries correctly. And any algorithm, even a "buggy one", shows the existence of an algorithm with the proved upper bound.

1.3 Our Contributions

Here are the paper's contributions, all of which were formalized using the EasyCrypt proof assistant:

- We formalize a general notion of list-based computational problems in the query model.
- We give a generic formalization of the adversarial method, expressing this as a two-party lower bound game between an algorithm and adversary.
- We formalize a two-party upper bound game for establishing upper bounds of algorithms, where the adversary adaptively and incrementally realizes an input to the algorithm.
- We prove a natural connection between the lower and upper bound games.
- We provide an axiom-free EasyCrypt theory for working with bounds involving logarithms. We also provide an axiom-free theory formalizing well-founded relations, induction and recursion.
- We demonstrate the utility and generality of our framework by applying it to three computational problems, giving evidence for why our apparently "passive" notion of algorithm is general enough to express all algorithms, including recursive ones.

1.4 Paper Outline

The rest of the paper is structured as follows. In Section 2, we give an introduction to EasyCrypt and detail the proof practices we follow. In Section 3: we describe our framework for formalizing computational problems, as well as the lower and upper bound games; we consider the proof of the theorem connecting these games; and we consider the proof of the lower bound theorem for the or function. In Section 4, we apply our framework to the problem of searching in an ordered list, proving identical lower and upper bounds (the upper bound is for the binary search algorithm). We also (Section 4.1) consider a general EasyCrypt theory for working with bounds involving logarithms. In Section 5, we apply our framework to the problem of sorting a list of distinct elements, proving lower and upper bounds (the upper bound is for the merge sort algorithm) that are close, but with a slight gap. We also mention our EasyCrypt theory of well-founded relations, induction and recursion. In Section 6, we consider related work. Finally, in Section 7 we consider directions for future work.

2 The EasyCrypt Proof Assistant

EasyCrypt [6] is a tactic-based proof assistant with several program logics allowing one to state and prove lemmas about a simple programming language, pWhile, with non-recursive procedures, while loops, and probabilistic assignments, and in which programs are structured using a simple module system designed for expressing games. Modules consist of procedures along with global variables that can be manipulated by those procedures. They can be parameterized by abstract modules implementing module types, and can later be applied to concrete modules implementing those types. EasyCrypt has:

- a Hoare logic for proving partial correctness of procedures;
- a probabilistic Hoare logic, pHL, for proving probabilistic facts about procedures;
- a probabilistic relational Hoare logic, pRHL, for proving relations between procedures;
- an "ambient" higher-order logic, which is used for connecting judgments of the program logics, as well as for giving mathematical definitions and proving lemmas about them.

Simple ambient logic goals may be solved using SMT solvers. Finally, EasyCrypt has theories, which collect definitions, modules and module types, axioms and lemmas. Theories may be parameterized by types and operators, and instantiating a theory with values for those types and operators is done by a process called cloning. All axioms involving those types and operators must then be proved to hold.[4] E.g., if a theory T includes the parameters

```
op n : int. (* n is an integer *)
axiom gt0_n : 0 < n. (* n is positive *)
```

and we clone T instantiating n with the expression k * 2, then we must prove that k * 2 is positive.

To explain the meaning of judgments in EasyCrypt's logics, suppose M.f (M is a module) is a procedure with parameter x : int returning a value of type bool, and N.g is a procedure with parameter y : int returning a value of type bool.

The Hoare logic judgment

```
hoare [M.f : 0 < x ⟹ res]
```

[4] EasyCrypt provides a facility for cloning but leaving some types and operators abstract, so axioms involving them are not proved, but we do not use this facility in our work.

says that if we start M.f in a memory &m in which the value of the parameter field x is positive, and this execution terminates with a memory &n, then the result value field res of &n will be true (this says nothing when the execution does not terminate). If, instead, we wrote

```
hoare [M.f : 0 < x ==> !res]
```

then we would be saying that the result value field will be false, not true (! is negation).

The executions of M.f from &m can be modeled as the branches of a tree that splits upon random assignments, where each arc is labeled with the (non-zero) probability of the selected value from the support of the distribution. EasyCrypt allows sub-distributions, and so the sum of these probabilities may be less than 1. In addition, each iteration of a while loop is marked by a single arc labeled with probability 1. These trees can be infinitely splitting, even though EasyCrypt's distributions are discrete (a distribution on an infinite type will be non-uniform), and they can have infinite branches, corresponding to non-termination. Each branch has an associated probability (the limit of the multiplication of all its probabilities), and the sum of the probabilities of all the branches may be less than 1. A *run* is a branch with non-zero probability. When M.f is non-probabilistic, its tree has a single run, which either terminates in a final memory or is infinite.

The pHL judgment

```
phoare [M.f : 0 < x ==> res] = p
```

says that if we execute M.f from an initial memory &m in which the value of x is positive, the probability that the execution terminates in a memory with a result value field that is true is p – i.e., p is the sum of the probabilities of all the branches of the running of M.f from &m terminating with memories whose result value fields are true. If we prove this, we can then use some ambient logic tactics to conclude

```
forall &m (a : int), 0 < a ⇒ Pr[M.f(a) @ &m : res] = p
```

which says that for all memories &m and positive integers a, the probability that running M.f in the updating of &m that gives x the value a terminates in a memory whose result value field is true is exactly p. If we set p to 1%r (the integer 1 converted to type real) this means every run of M.f(a) from &m terminates in a memory whose result value field is true, and the sum of the probabilities of those runs is 1. And if M.f isn't probabilistic, its single run will terminate in a memory with a true result value field. This allows us to express total correctness. A judgment

```
phoare [M.f : true ==> true] = 1%r
```

says that M.f is *lossless*: no matter what memory M.f is started in (the memory includes its argument), it terminates with probability 1. This mean that any infinite branch (if there are any) has probability 0, i.e., all runs terminate, and the sum of the runs' probabilities is 1. If M.f is non-probabilistic, losslessness means that the single branch of M.f from any initial memory terminates.

The pRHL judgment

```
equiv [M.f ~ N.g : x{1} = y{2} ==> res{1} ⇒ res{2}]
```

says that if &1 and &2 are memories in which the value of x in &1 is equal to the value of y in &2, then the distributions on memories corresponding to running M.f and N.g on &1 and &2, respectively, satisfy the lifting of the formula res{1} ⇒ res{2} (if the result of the first memory is true, then the result of the second memory is true) to pairs of memory distributions. Given a proof of this judgment, we can then prove the ambient logic formula

Listing 1 Framework Parameters.

```
type inp, out, aux. op univ : inp list. op arity : {int | 0 ≤ arity} as ge0_arity.
axiom univ_uniq : uniq univ.
op good : aux → inp list → bool. op f : aux → inp list → out option.
axiom good (aux : aux, inps : inp list) :
  size inps = arity ⇒ all (mem univ) inps ⇒ good aux inps ⇒ exists (y : out), f aux inps = Some y.
axiom bad (aux : aux, inps : inp list) :
  size inps ≠ arity ∨ ! (all (mem univ) inps) ∨ ! good aux inps ⇒ f aux inps = None.
```

```
forall &m (a : int), Pr[M.f(a) @ &m : res] ≤ Pr [N.g(a) @ &m : res]
```

Alternatively, if we prove the judgment

```
equiv [M.f ~ N.g : x{1} = y{2} ⟹ res{1} = res{2}]
```

we can conclude

```
forall &m (a : int), Pr[M.f(a) @ &m : res] = Pr [N.g(a) @ &m : res]
```

2.1 Our Proof Practices

Unrestricted use of SMT solvers can lead to poorly documented, unstable proofs. We counter this as follows. Sometimes we avoid using SMT solvers entirely, using the EasyCrypt directive that disallows the smt tactic. More commonly, we use the directive requiring that every goal solved using the smt tactic be solved by *both* the Z3 and Alt-Ergo solvers. And in every use of smt, we explicitly list the lemmas the solvers may use (in addition to their internal theories). smt() means to only use the solvers' internal theories.

3 EasyCrypt Framework for Lower and Upper Bounds

In this section we describe our framework – defined as a theory Bounds in EasyCrypt (788 lines of code) – for describing computational problems and proving lower and upper bound theorems, using lower and upper bound games. We also consider the proof of a theorem connecting the lower and upper bound games. And we consider the proof of the lower bound theorem for the or function that was sketched in the introduction.

3.1 Expressing Computational Problems

The Bounds theory is parameterized by several types and operators (functions) that allow us to express computational problems. They are described in Listing 1. An *input list* is a list of values of size arity (a non-negative integer; ge0_arity is introduced as the name of an axiom saying that 0 ≤ arity) where each value has type inp (the *input type*) and is an element of a finite universe univ (a list of distinct (unique) elements). The type aux is a type of *auxiliary values*, and a value of type aux is the first parameter of the good and f operators. When inps is an input list, good aux tests whether it is a *good* input list for our computational problem, *according to* the auxiliary value aux. The operator f represents our *computational problem*; f aux inps returns an optional value of the *output type*, out. We axiomatize that:

- (good) If inps is an input list that is good according to aux, then f aux inps returns Some of some output value.
- (bad) Otherwise, f aux inps returns None.

When we instantiate the above types and operators, we will typically leave arity *abstract*, so that algorithms and adversaries will have to work for all input list sizes.

Below are three examples of how computational problems can be formalized using our framework. We believe many more problems can be easily encoded using variations of these techniques.

3.1.1 Or Function

If our computational problem is the or (disjunction) function, we can instantiate inp, univ, out and aux to bool, [true; false], bool and unit (whose only value is ()), where good () always returns true (all input lists satisfy it). When inps is an input list, f inps returns Some true, if at least one of the elements of inps is true, and Some false, otherwise. We use this instantiation of Bounds in Section 3.6.

3.1.2 Searching in an Ordered List

To see why we parameterize f and good by an auxiliary value, consider the problem of searching in an ordered list inps of elements from a finite range of at least two integers for a given value k that is guaranteed to occur at least once in inps, returning the first index (indices are in the range $0, \ldots,$ arity $- 1$) into inps where k is found. Here we can instantiate inp and out to int, and instantiate univ to the finite range of integers. But both good and f need to know what k is, and so we set aux to int, so the first arguments to good and f can be k. Then an input list inps is satisfied by good k iff it is sorted in (not necessarily strictly) ascending order and contains at least one occurrence of k, and f k inps returns Some of the first index into inps where k is found, when inps is a good input list relative to k, and returns None, otherwise. We use this instantiation of Bounds in Section 4.

3.1.3 Sorting

We can encode the problem of sorting a list of distinct elements of size len (at least 1) according to some total ordering, as follows. First, we instantiate inp and univ to bool and [true; false], and instantiate arity to len $*$ len. Thus an index into an input list inps can be thought of as encoding a pair (i, j), where i and j are indices into the list of distinct elements, and asking for the boolean corresponding to (i, j) can be thought of as querying whether the ith element of the list of distinct elements is less-than-or-equal to the jth element. In this example, we don't need an auxiliary value, and so aux can be unit. The predicate good, though, should test whether an inps satisfies the properties of a total ordering. Finally, the operator f should transform a list of booleans representing a total ordering to Some of the permutation of the indices of the list of distinct elements that is sorted according to the total ordering, and return None when not given a total ordering. Thus out can be int list (even though the len! permutations on the indices $0, \ldots,$ len $- 1$ are a small subset of this type). We use this instantiation of Bounds in Section 5, when our lower and upper bounds will be expressed in terms of len (not arity).

3.2 Adversaries

In the following two sections, we will consider two sub-theories of Bounds: LB for lower bounds, and UB, for upper bounds. Both of these theories share the same kind of *adversary*, defined by the following module type (interface):

```
module type ADV = { proc *init() : aux   proc ans_query(i : int) : inp }.
```

Listing 2 Lower Bound Game.

```
1  module G(Alg : ALG, Adv : ADV) = {
2    proc main() : bool * int = {
3      var inpss : inp list list; var don : bool; var error : bool;
4      var stage : int; var queries : int fset; var aux : aux;
5      var i : int; var inp : inp;
6      aux <@ Adv.init(); Alg.init(aux);
7      inpss ← init_inpss aux; error ← false; don ← inpss_done aux inpss; stage ← 0; queries ← fset0;
8      while (!don ∧ !error) {
9        i <@ Alg.make_query();
10       if (0 ≤ i < arity ∧ ! i \in queries) {
11         queries ← queries `|` fset1 i; stage ← stage + 1;
12         inp <@ Adv.ans_query(i);
13         Alg.query_result(inp);
14         inpss ← filter_nth inpss i inp; don ← inpss_done aux inpss;
15       }
16       else { error ← true; }
17     }
18     return (error, stage);
19   }
20 }.
```

A module of this type may have global (private) variables, and implements at least the two procedures of the module type. There may be auxiliary procedures, and all procedures may access the global variables. The procedure init takes no arguments and returns an auxiliary value – because in both the lower and upper bound games we let the adversary choose the auxiliary value. The asterisk before its name specifies that init must initialize the module's global variables. The procedure ans_query will only be called with an index i into an input list, i.e., an integer between 0 and arity − 1. It is a request for the value of the ith element of the input list. In addition to returning this value, a stateful adversary will update some of its global variables, so it knows how to properly respond to subsequent queries.

3.3 Lower Bounds Subtheory

In this subsection, we consider the lower bounds subtheory, LB. A *lower bound algorithm* is a module satisfying the interface:

```
module type ALG = { proc *init(aux : aux) : unit   proc make_query() : int
  proc query_result(x : inp) : unit }.
```

Its init procedure is given the auxiliary value chosen by the adversary, and initializes its global variables. The procedure make_query asks the algorithm to issue a query, which should be an index into an input list, i.e., an integer in the range $0, \ldots,$ arity $- 1$. And the procedure query_result is called to tell the algorithm the adversary's answer to the algorithm's last query. Note that the algorithm does not report an answer to the computational problem.

The *lower bound game*, G, is defined in Listing 2. G is parameterized by an algorithm Alg and adversary Adv, i.e., modules with the module types ALG and ADV, respectively. It defines a procedure main that takes no arguments, and returns a value of type bool * int. The boolean indicates whether the game ended with the algorithm committing an *error* by issuing an illegal query, and the integer is the *stage* (number of steps executed) at the game's termination. main uses two groups of local variables (which are not accessible to Alg and Adv). The ones on lines 3–4 are persistent; they are initialized before the game's while loop, and are updated by each loop iteration. The ones on line 5 are temporary variables; their values are reset at each iteration. main begins (line 6) by asking Adv to initialize itself and choose the auxiliary value, which is stored in aux, and then asking Alg to initialize itself, letting it know the auxiliary value. On line 7, main initializes the rest of the persistent variables:

- inpss records (as a list of lists with no duplicates) the good inputs lists (relative to aux) that are consistent with the queries issued so far and their answers. It is initialized to all the good input lists, using an operator init_inpss.
- error records whether the algorithm has issued an illegal query.
- don records whether all elements of inpss have the same answer to the computational problem (relative to aux), i.e., whether all the elements of the result of mapping f aux over inpss are the same (this is what the operator inpss_done tests). Depending upon good, f and aux, this may be true initially; and it's true whenever inpss is empty.
- stage records the current stage, and is initialized to 0.
- queries records the set of queries that have been issued by the algorithm, and is initialized to the empty set.

The while loop of main runs as long as both error and don are false, i.e., as long as the algorithm has not committed an error and at least two elements of inpss have different answers. An iteration of the while loop begins by asking (line 9) Alg to issue a query, which is stored in i. If (test on line 10) i is not a good index into an input list or repeats an earlier query, this causes error to be set to true. Otherwise (line 11) the set of queries is updated to include i and the stage is incremented. Adv (line 12) is then asked to answer the query i, producing inp. On line 13, Alg is informed of the adversary's answer. On line 14, inpss is filtered to only retain input lists whose ith elements are equal to inp, and then don is recomputed based on the updated inpss, seeing if the game is now done. Once the while loop has terminated (line 18), because either error or don became true, main returns the final error status and stage.

If Adv answers a query in a way that is inconsistent with inpss, i.e., so that after filtering, inpss becomes empty, then don will be set to true, and so the game will end without error. It doesn't matter whether Alg notices this inconsistency upon the final call to Alg.query_result. But at every call to Alg.make_query, Alg can be assured that all of the answers it has received so far are consistent.

When proving a lower bound theorem for a given computational problem, we program a concrete adversary Adv whose procedures are provably lossless, and then prove that for all algorithms Alg whose procedures are lossless and don't read or write the global variables of Adv (or vice versa),

```
Pr[G(Alg, Adv).main() @ &m : res.`1 ∨ ϕ(arity) ≤ res.`2] = 1%r.
```

holds, where ϕ is the desired function of arity. We can read the conclusion as saying that, when started in a memory &m (which the game doesn't depend on, because G has no global variables, and Adv.init and Alg.init initialize the adversary's and algorithm's global variables), with probability 1, the game terminates either with the algorithm having committed an error (by asking an out of range query or the same query twice), or with the final stage being at least ϕ(arity).

An algorithm that issues duplicate queries can be converted into a more efficient one by caching query answers, and so from the point of view of lower bounds, treating query-repeating algorithms as erroneous is sound. Our motivation for signaling an error when a query is repeated is to ensure the game terminates – which is technically useful.

When proving that the procedures of an algorithm (or adversary) are lossless, it is sometimes necessary to condition this on a *termination invariant* on the global variables of the algorithm (or adversary). When doing this, we show that the initialization procedures establish this invariant with probability 1, and that the other procedures preserve the invariant with probability 1. We will see this in action when we consider the proof of the upper bound theorem for the merge-sort algorithm (Section 5.2).

Because our conclusion is with probability 1 and we assume the procedures of the algorithm are lossless, we can apply a generic lemma that we have proved using pHL to show that G(Alg, Adv).main is lossless. And we can use this fact to reduce our theorem to a "main lemma" whose conclusion is an ordinary Hoare (partial correctness) judgment:

```
hoare [G(Alg, Adv).main : true ==> res.`1 \/ φ(arity) <= res.`2].
```

The challenging part of proving the main lemma is handling the game's loop. In Hoare logic, this is done using a loop invariant, and we need a loop invariant that is true when the loop is first entered, is preserved by the loop, and where the conjunction of the loop invariant and the fact that the game has ended – and so either the algorithm has committed an error, or f aux agrees on all elements of inpss – tells us that if the algorithm hasn't committed an error, then the game has run long enough to give us the desired lower bound. In Sections 3.6, 4.2 and 5.1, we will see three rather different examples of loop invariants supporting lower bound proofs.

3.4 Upper Bounds Subtheory

In this subsection, we consider the upper bounds subtheory, UB. In order to define the module type of upper bound algorithms, we need a datatype response of algorithm responses, along with some associated operators:

```
type response = [ Response_Query of int | Response_Report of out ].
op dec_response_query (resp : response) : int option =
  with resp = Response_Query i ⇒ Some i  with resp = Response_Report _ ⇒ None.
op dec_response_report (resp : response) : out option =
  with resp = Response_Query _ ⇒ None  with resp = Response_Report x ⇒ Some x.
op is_response_query (resp : response) : bool = dec_response_query resp ≠ None.
op is_response_report (resp : response) : bool = dec_response_report resp ≠ None.
```

A value of type response either has the form Response_Query i, for an integer i, representing a query by the algorithm, or the form Response_Report x, for a value x of type out, representing the algorithm's reporting of an answer, x, to a computational problem. The operator dec_response_query tries to decode a value of type response as a query, yielding Some i when given Response_Query i, and yielding None when given a value with constructor Response_Report. And dec_response_report is similar, but with the constructors swapped, and so producing an optional value of type out. The operators is_response_query and is_response_report use these operators to test whether a value of type response has constructor Response_Query or Response_Report.

An *upper bound algorithm* is a module satisfying the interface:

```
module type ALG = { proc *init(aux : aux) : unit   proc make_query_or_report_output() : response
  proc query_result(x : inp) : unit }.
```

The procedures init and query_result are just like the procedures of the same names of a lower bound algorithm. And make_query_or_report_output asks the algorithm to either issue another query or report the answer to the computational problem.

The *upper bound game*, G, is defined in Listing 3. This game is similar to the lower bound game (Listing 2), and so we focus on the differences. The interpretations of error and don are different:

- error can still become true because Alg issues an illegal query, but also because the final answer it reports is incorrect.
- don can become true because Alg reports the correct answer, but also because Adv chooses aux or answers queries in a way that causes inpss to become empty.

Listing 3 Upper Bound Game.

```
1  module G(Alg : ALG, Adv : ADV) = {
2    proc main() : bool * int = {
3      var inpss : inp list list; var aux : aux; var error : bool;
4      var don : bool; var stage : int; var queries : int fset;
5      var resp : response; var i : int; var inp : inp; var out : out;
6      aux <@ Adv.init(); Alg.init(aux);
7      inpss ← init_inpss aux; error ← false; don ← inpss = []; stage ← 0; queries ← fset0;
8      while (!don ∧ !error) {
9        resp <@ Alg.make_query_or_report_output();
10       if (is_response_query resp) {
11         i ← oget (dec_response_query resp);
12         if (0 ≤ i < arity ∧ ! i \in queries) {
13           queries ← queries `|` fset1 i; stage ← stage + 1;
14           inp <@ Adv.ans_query(i);
15           Alg.query_result(inp);
16           inpss ← filter_nth inpss i inp; if (inpss = []) { don ← true; }
17         }
18         else { error ← true; }
19       }
20       else {
21         out ← oget (dec_response_report resp);
22         if (inpss_answer aux inpss out) { don ← true; }
23         else { error ← true; }
24       }
25     }
26     return (error, stage);
27   }
28 }.
```

On line 7, don is initialized to be true iff inpss is empty. On line 9, Alg is asked to either issue a query or report the final answer, encoded in a value resp of type response. If resp encodes a query, then it is decoded on line 11, resulting in the query i. Lines 12–15 and line 18 are the same as lines 10–13 and line 16 of the lower bound game. But line 16 is different: it sets don to true if inpss has become empty. Alternatively, if resp encodes the reporting of an output, then line 21 decodes resp to out. Line 22 then checks (via the operator inpss_answer) whether out is the answer (relative to aux) for all the elements of inpss, i.e., whether every element of the result of mapping f aux over inpss is equal to Some out. If the answer is "yes", then don is set to true; otherwise error is set to true.

When proving an upper bound theorem for algorithm Alg, we prove that Alg's procedures are lossless, and that, for all adversaries Adv whose procedures are lossless and don't read or write the global variables of Alg, that

```
Pr[G(Alg, Adv).main() @ &m : ! res.`1 ∧ res.`2 ≤ φ(arity)] = 1%r.
```

holds, where ϕ is the desired function of arity. We can read the conclusion as saying that, when started in a memory &m, with probability 1, the game terminates without the algorithm having committed an error (and so with Alg having reported the correct answer to the computational problem, unless Adv caused inpss to become empty) and with the final stage being no more than ϕ(arity).

Because our conclusion is with probability 1 and we assume the procedures of the adversary are lossless, we can apply a generic lemma that we have proved using pHL to show that G(Alg, Adv).main is lossless. And we can use this fact to reduce our theorem to a "main lemma" whose conclusion is an ordinary Hoare judgment:

```
hoare [G(Alg, Adv).main : true ⟹ ! res.`1 ∧ res.`2 ≤ φ(arity)].
```

Listing 4 Conversion from Upper to Lower Bound Algorithm.

```
module UBAlg_to_LBAlg (UBAlg : UB.ALG) : LB.ALG = {
  proc init(aux : aux) : unit = { UBAlg.init(aux); }
  proc make_query() : int = {
    var resp : UB.response; var i : int; resp <@ UBAlg.make_query_or_report_output();
    if (UB.is_response_query resp) { i ← oget (UB.dec_response_query resp); } else { i ← −1; }
    return i;
  }
  proc query_result(x : inp) : unit = { UBAlg.query_result(x); }
}.
```

In Sections 4.3 and 5.2 we will see two rather different examples of loop invariants supporting upper bound proofs.

If Adv answers a query inconsistently, so that after filtering, inpss becomes empty, and thus don is set to true and the game ends without error, the final call to Alg.query_result must still update the state of Alg in such a way that the loop invariant is preserved. But at every call to Alg.make_query_or_report_output, Alg can be assured that all of the answers it has received so far are consistent.

A meta-level analysis using the semantics of EasyCrypt shows that if Alg loses a run of the upper bound game against Adv and for bound ub (the only run if both Alg and Adv are non-probabilistic), there is an auxiliary value aux, and an inps that is good relative to aux such that there is a run (the only run if Alg is non-probabilistic) of Alg against the non-probabilistic, non-adaptive adversary that picks aux and then answers queries according to inps in which Alg loses. Thus if we are able to prove that Alg wins the upper bound game against all such *hard-coded* adversaries with probability 1, it will actually win the game against all adversaries with probability 1. We haven't tried to prove this meta result in EasyCrypt, and we don't assume it in our proofs.

3.5 Connections between Frameworks

Listing 4 defines a parameterized module UBAlg_to_LBAlg that converts an upper bound algorithm to a lower bound one. The only issue is what to do when the upper bound algorithm reports an answer to the computational problem – an action that isn't allowed for a lower bound algorithm. UBAlg_to_LBAlg translates the reporting of an answer into the illegal query −1. (This is so that, if the upper bound algorithm reports an answer before there is a unique answer according to the remaining consistent input lists, and so the upper bound game terminates with an error, the same thing will happen in the lower bound game.)

Through a sequence of lemmas using pRHL, Hoare logic, pHL and the ambient logic, we were able to prove the theorem saying that for all integers lb and ub, upper bound algorithms Alg whose procedures are lossless, adversaries Adv whose procedures are lossless and don't read/write the global variables of Alg, and memories &m:

```
Pr[LB.G(UBAlg_to_LBAlg(Alg), Adv).main() @ &m : res.`1 ∨ lb ≤ res.`2] = 1%r ⇒ ub < lb ⇒
Pr[UB.G(Alg, Adv).main() @ &m : res.`1 ∨ ub < res.`2] = 1%r.
```

Thus if we have proved a lower bound theorem involving Adv and lb, and we know ub < lb, we can conclude that every run of the upper bound game between Alg and Adv (when both Alg and Adv are non-probabilistic there is only one run) ends with either Alg committing an error or the game having run strictly more than ub steps, and that the sum of the probabilities of those runs is 1. This of course implies that we cannot prove an upper bound theorem for Alg with bound ub.

3.6 Lower Bound for Or Function

As a warm up exercise, we formalized the proof from Section 1.1 that any algorithm computing the or function on a list of booleans of size arity must query every element of the list, in the worst case. We clone our general Bounds framework as described in Section 3.1.1. Our adversary is stateless and answers all queries with false. Our loop invariant for the lower bound game's while loop simply says (in addition to some housekeeping properties) that inpss is all the input lists in which all the queried indices are false. The formalization of the computational problem and the subsequent lower bound proof took 324 of code.

4 Application to Searching in an Ordered List

In this section, we consider proofs of lower and upper bound theorems for the computational problem of searching in an ordered list of integers (coming from a finite range of at least two elements) of size arity in which an element k occurs at least once, returning the first index into the list where k can be found. We clone our general Bounds theory as described in Section 3.1.2.

4.1 Theory for Reasoning about Bounds Involving Logarithms

We have developed a reusable (also used in Section 5) EasyCrypt theory IntLog (1440 lines of code) for reasoning about bounds involving integer logarithms. We have the operators

```
op (%/) : int → int → int. op (%%/) : int → int → int.
```

%/ is EasyCrypt's integer division operator, and we define %%/ to add one to the result of n %/ b when n is not divisible by b. Then we can prove that, for all integers n and $b \geq 1$, n %/ $b = \lfloor n/b \rfloor$[5] and n %%/ $b = \lceil n/b \rceil$. We define integer logarithm operators rounding down and up

```
op int_log : int → int → int. op int_log_up : int → int → int.
```

Then we prove that, for all $b \geq 2$ and $n \geq 1$, int_log b $n = \lfloor \log b\ n \rfloor$ and int_log_up b $n = \lceil \log b\ n \rceil$, where log is the real number logarithm operator. For use in lower and upper bound proofs, it is convenient to define operators

```
op divpow2 (n k : int) : int = n %/ (2 ^ k). op divpow2up (n k : int) : int = n %%/ (2 ^ k).
```

where ^ is exponentiation. We will mention some of the many lemmas we have proved about these operators in the following sections.

4.2 Lower Bound Proof

Our adversary is expressed in terms of the minimum element of the universe, a, and a + 1, which we call b. Its init procedure chooses b as the auxiliary value – the value the algorithm must search for. The adversary has global variables win_beg and win_end of type int and win_empty of type bool that determine the current *window of uncertainty*, where win_beg and win_end are indices into an input list such that win_beg ≤ win_end, and if win_empty holds (the window is empty), then win_beg = win_end and win_end < arity − 1 (the window does not end at the end of the input list). The *window size* (computed by operator win_size)

[5] In EasyCrypt's syntax, one actually must write floor (n%r / b%r).

is defined to be: 0, if the window is empty; and win_end − win_beg + 1, otherwise. The procedure init initializes win_beg to 0, win_end to arity − 1 and win_empty to false. When ans_query is called with query i, it acts as follows:

- if the window is empty (win_size returns 0), it returns witness – some unknown but fixed value[6];
- else, if i is strictly smaller than win_beg, it returns a;
- else, if i is strictly greater than win_end, it returns b;
- else, if the window has size 1 and win_end is arity − 1, it returns b;
- else, if the window has size 1 and win_end is strictly less than arity − 1, it sets win_empty to true and returns witness;
- else, if $i <$ (win_beg + win_end) %%/ 2, it sets win_beg to $i + 1$ and returns a;
- else it sets win_end to $i - 1$ and returns b.

The part of the game's loop invariant relating to inpss says that, if the window is not empty, then:

- for all i between win_beg and win_end, inclusive, the input list consisting of a's up to but not including position i, and then b's thereafter, is in inpss; and
- if win_end < arity − 1, the input list consisting of a's up to and including position win_end, and then b's thereafter, is in inpss.

The bound part of the loop invariant says that:

```
(win_end = arity − 1 ⇒
 divpow2up arity stage ≤ win_size win_empty win_beg win_end) ∧
(win_end < arity − 1 ⇒
 divpow2 arity stage ≤ win_size win_empty win_beg win_end)
```

We are able to prove that:

- if the window is not empty and the game is finished, then the window size is 1; and
- if win_end < arity − 1 and the game is finished, then the window is empty.

At the game's beginning, win_end = arity − 1. If this is still true at the game's end, it follows that the window is not empty and divpow2up arity stage is 1, and thus – by one of the lemmas of IntLog – that int_log_up 2 arity ≤ stage. Otherwise, there is a point at which win_end is set to one less than the algorithm's query, i. At this point, we *don't* know that the new window size is at least the old window size divided by two, rounding *up*, but it *is* at least the old window size divided by two, rounding *down*. And thus we switch to only knowing divpow2 arity stage ≤ win_size win_empty win_beg win_end. Then, when the game ends, we have that the window is empty, so that divpow2 arity stage is 0. Because it is 0, we can conclude – using another of our IntLog lemmas – that int_log_up 2 arity ≤ stage. Consequently, our lower bound theorem has bound int_log_up 2 arity.

4.3 Upper Bound Proof

Because binary search can be expressed iteratively, it was straightforward to define the binary search algorithm in our framework. Its state consists of the element aux to be searched for, along with input list indices low and high, where low ≤ high and the following invariant holds:

- every element of inpss is sorted and has at least one occurrence of aux between positions low and high inclusive;
- in every element of inpss, there are no occurrences of aux at positions strictly less than low;

[6] Used to document that the proof doesn't depend on the value returned.

- no indices between low and high inclusive have been asked as queries.

low and high are initialized to 0 and arity − 1. The *window size* is defined to be high − low + 1. When make_query_or_report_output is called:

- if the window size is 1, the algorithm reports its answer, low;
- else, the algorithm queries the midpoint mid ← (low + high) %/ 2.

When query_result is called with the answer, x:

- if $x <$ aux, it sets low to mid + 1;
- else, it sets high to mid.

In either case, the new window size is no more than the old window size divided by 2, rounding *up*. Thus the bound part of the loop invariant can be

```
stage ≤ int_log_up 2 arity ∧ win_size low high ≤ divpow2up arity stage
```

When the window size is 2 or more, one of our IntLog lemmas tells us that stage < int_log_up 2 arity, ensuring the bound invariant will be preserved as stage is incremented. Our overall upper bound is thus int_log_up 2 arity.

4.4 Conclusions

The formalization of the searching problem and some associated lemmas took 148 lines of code. And the proof of the lower bound (resp., upper bound) theorem took 724 (resp., 353) lines of code.

Our upper and lower bounds are identical, proving that the binary search algorithm is optimal. Our first version of the lower bound theorem used a simpler approach but only achieved a bound of int_log 2 arity. We noticed this was not optimal for the arity of 3, and then wrote an OCaml program[7] to generate optimal strategies for larger arities and small universes. The program's results helped us develop an adversarial strategy supporting our tighter lower bound theorem.

5 Application to Sorting

In this section, we consider proofs of lower and upper bound theorems for the computational problem of sorting a nonempty list of distinct elements of size len. We clone our general Bounds theory as described in Section 3.1.3.

5.1 Lower Bound Proof

Our adversary has a single global variable inpss of type inp list list consisting of its own copy of the list of consistent input lists maintained by the lower bound game. Initially, this is init_inpss (), which has len! elements – because the set of all permutations of the indices $0, \ldots,$ len $- 1$ is in one-to-one correspondence with all the total orderings on those indices. When its ans_query procedure is called with a query q encoding a comparison query (i, j) (and so asking if element i of the list of distinct elements is less-than-or-equal-to element j), it partitions inpss into two lists:

- inpss_t is all the elements of inpss in which position q is true; and
- inpss_f is all the elements of inpss in which position q is false.

[7] All OCaml programs are included in our repository.

If the size of inpss_f is at least as big as the size of inpss_t, it sets inpss to inpss_f and returns false; otherwise it sets inpss to inpss_t, and returns true. Because the new size of inpss in each step is at least the old size divided by 2, rounded *up*, the initial size of inpss is fact len, and the game isn't over until inpss contains a single element, it is straightforward to prove a lower bound of int_log_up 2 (fact len). We lower-approximate this in two ways: (1) (len ∗ int_log 2 len) %/ 2; and (2) len ∗ (int_log 2 len) − 2 ∗ 2^(int_log 2 len). When len is at least 11, we prove (2) ≥ (1). For example, when len is 20,000, (2) evaluates to 247,232 whereas int_log_up 2 (fact len) evaluates to 256,909, and so the gap is only 9,677 comparisons.

5.2 Upper Bound Proof

Our upper bound result is for the merge sort algorithm. First, we defined a recurrence wc returning (what we later prove is) an upper-approximation of the worst-case number of comparisons needed by merge sort when sorting a nonempty list of distinct elements of size n:

$$\mathsf{wc}\, n = \begin{cases} 0, & \text{if } n = 1, \\ \mathsf{wc}\,(n\,\%/\,2) + \mathsf{wc}\,(n\,\%\%/\,2) + n - 1, & \text{otherwise.} \end{cases}$$

If we had made a mistake in defining wc, our upper bound proof would have failed. We were able to upper-approximate wc len as len ∗ int_log 2 len.

In order to define wc in EasyCrypt and prove the necessary properties about it, we used our axiom-free theory WF (503 of code) for well-founded relations, induction and recursion.[8] In this theory, we mirrored the set theoretic formalization of these concepts within EasyCrypt's higher-order logic, using the type

```
type 'a rel = 'a → 'a → bool.
```

to represent relations. This theory has now been added to the official EasyCrypt library.

Because merge sort is a recursive algorithm, we opted to let Alg have a single global variable term of type term where term is the inductive datatype

```
type term = [ Sort of int list | List of int list | Cons of int & term | Merge of term & term
  | Cond of int & int & int list & int list ].
```

We think of the elements of term as *terms* of an ad hoc functional programming language. A term is evaluated relative to a total ordering on indices. Sort xs evaluates to the result of sorting xs. List xs evaluates to xs. Cons i t is evaluated by evaluating t to xs, and then returning $i :: xs$. Merge t u is evaluated by first evaluating t to xs, next evaluating u to ys, and then returning the result of merging xs and ys. Finally, Cond i j us vs queries whether $i \leq j$ in the total ordering, returning: $i :: zs$, where zs is the result of merging us and $j :: vs$, when the answer is "yes"; and $j :: zs$, where zs is the result of merging $i :: us$ and vs, when the answer is "no".

The procedure init initializes term to Sort $[0; \ldots; \mathsf{len} - 1]$. We implemented a single-step operational semantics that runs a term until it either produces an answer (List xs) which can be reported to the algorithm (as the result of make_query_or_report_output) or asks a query (Cond), which can be returned to the game, recording the blocked term in the global variable term. Once the answer to the query is supplied via query_result, the algorithm applies the answer to term, so execution can continue upon the next call to make_query_or_report_output. To ensure termination of make_query_or_report_output, we make use of a well-formedness invariant on terms together with a termination metric.

[8] We developed this theory for another purpose, but it has not been referenced in the literature, to-date.

Our loop invariant says that:

- term is well-formed;
- for each remaining consistent input list inps (representing a total ordering), the evaluation of term relative to inps is the same as the result of sorting $[0; \ldots; \mathsf{len} - 1]$ using inps;
- wc_term term + stage $\leq$ wc len, where wc_term recursively upper-bounds the numbers of comparisons term can make in the worst case; and
- all the comparisons term could potentially make have not yet been issued as queries.

At the start of the game, wc_term term is wc len and stage is 0. As the game progresses, wc_term term becomes smaller as stage becomes bigger. When the game ends, wc_term term is 0, and so we get that stage is no more than wc len, which we know is no more than len $*$ int_log 2 len.

5.3 Conclusions

The formalization of the sorting problem took 557 lines of code (this included the formalization of total orderings as lists of booleans). The lower bound proof took 547 lines of code, which included 59 lines of generic code for showing that two lists of unique elements have the same size using the existence of a bijection between them. And the upper bound proof took 1740 lines of code, much of which consisted of meta-theoretic results about the ad hoc functional language used to model suspendable, recursive computations.

Our lower and upper bounds are close, but there is a small gap. E.g., if we compare int_log_up 2 (fact len) (the tightest form of the lower bound) with wc len (the tightest form of the upper bound) when len is equal to 20,000, we obtain 256,909 and 267,233, respectively, for a gap of only 10,324. We are considering how the gap could be closed.

6 Related Work

Formalization of lower and upper bounds. There is existing work [14, 20, 11, 1, 17] formalizing specific lower bounds using proof assistants in computational models other than the query model. Some of these works include reusable libraries applicable to certain classes of problems. General frameworks for certifying upper bounds were developed in [12, 18], but again not in the query model. Eberl [11] and Azevedo de Amorim [1] have formalized lower bounds on comparison-based sorting problems in Isabelle and Coq, respectively. Both works utilize decision trees, even though the two recursive datatypes representing them are rather different. Their proofs give a lower bound, in terms of the number of permutations on list indices, on the heights of decision trees, and then lower-approximate this bound in terms of list size. In our work we formalize a similar lower bound as an instance of our framework, showing that lower bound games and adversarial reasoning are good and flexible abstractions. In fact, the decision tree model and the query model are closely related, and we expect that a formalization of the adversarial approach could also be based on decision trees. We leave the study of the relations between the two formalizations to future work. In contrast to the formalizations of Eberl and Azevedo de Amorim, our work is much more general. Our generic notions of algorithms and adversaries, and generic definitions of lower and upper bound games apply uniformly to all computational problems. Any algorithm can give rise to an upper bound theorem, assuming the proof goes through; similarly, any adversary supports a lower bound theorem, assuming the proof goes through. We are not aware of other works that have formalized the query model directly. Recent work by Tassarotti et al. [22] formalizes probably approximately correct (PAC) learnability in Lean. Classifiers, in the PAC learning model, are similar to queries in the query model. It would be interesting to explore if our framework could be used in this model.

Game-based formal reasoning. Extensive research has employed formal reasoning based on games, often in the context of cryptographic security. Novak [19] developed a framework for proving the security of cryptographic schemes based on probabilistic games in Coq. Barthe et al. [7] formalized a framework in Coq with similar goals where cryptographic schemes and protocols are modeled by probabilistic games, parameterized by adversarial entities. Subsequent work in this direction led to the design of EasyCrypt [8, 6], which provides better support for this kind of reasoning. We use EasyCrypt in our work, demonstrating its utility for game-based reasoning beyond security. An Isabelle/HOL framework for proving the security of cryptographic schemes based on probabilistic games has recently been developed [9]. Recent work [5] has extended EasyCrypt with a cost model, allowing proofs about execution time and numbers of oracle calls and restrictions on the cost of adversaries. It would be interesting to investigate if this extension could be used to reason about lower bounds in the query model in a more natural way. Game-based reasoning supported by a proof assistant has also been used for certifying results useful in computer networking [4]. The recent work of [10] has employed a two-player game between an adversary and an algorithm to certify lower bounds for the online bin stretching problem. The proposed game, used to establish lower bounds for varying numbers of bins, is formalized in Coq. Interestingly, this work also proves some previously unknown lower bounds. As a comparison, our framework targets a larger class of computational problems in the query model. Moreover, it is not limited to lower bounds, but also explores upper bounds and the connection between them.

7 Future Work

In future work, we would like to generalize our framework for defining computational problems in various ways. It would be nice to be able to directly model problems whose inputs are not lists, e.g., graphs, instead of having to encode them using lists. We would also like to be able to model problems where answers are not required to be unique, e.g., searching for any element of a list satisfying some predicate.

The formalization of the ad hoc functional language we used to model suspendable, recursive query-model algorithms in our upper bound proof for merge sort (Section 5.2) could be generalized so as to be applicable to recursive algorithms in general, instead of just merge sort. Proving the meta-theoretic results once and for all in a more general form would save a great deal of effort when formalizing upper bound theorems for other algorithms.

We would like to extend our framework to be able to handle randomized algorithms that are allowed to produce incorrect results with small probabilities. In such an extended framework, the probabilistic nature of EasyCrypt could be an asset.

Finally, we would like to use our existing framework or future versions of our framework to discover new lower or upper bounds, as opposed to simply formalizing variations of standard results in our framework(s), as we have done in the current work.

References

1 Arthur Azevedo De Amorim. Poleiro: Analyzing Sorting Algorithms, 2015. URL: `http://poleiro.info/posts/2015-03-25-analyzing-sorting-algorithms.html`.

2 Sepehr Assadi. Advanced Algorithms II: Sublinear Algorithms: Lecture 3, 2020. URL: `https://people.cs.rutgers.edu/~sa1497/courses/cs514-s20/lec3.pdf`.

3 Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis (third edition)*. Pearson/Prentice Hall, 2000.

4 Alexander Bagnall, Samuel Merten, and Gordon Stewart. A library for algorithmic game theory in Ssreflect/Coq. *J. Formaliz. Reason.*, 10(1):67–95, 2017.

5 Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. Mechanized proofs of adversarial complexity and application to universal composability. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15–19, 2021*, pages 2541–2563. ACM, 2021.

6 Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer International Publishing, 2014.

7 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21–23, 2009*, pages 90–101. ACM, 2009.

8 Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO 2011, pages 71–90. Springer-Verlag, 2011.

9 David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2):494–566, 2020.

10 Martin Böhm and Bertrand Simon. Discovering and certifying lower bounds for the online bin stretching problem. *CoRR*, abs/2001.01125, 2020. `arXiv:2001.01125`.

11 Manuel Eberl. Lower bound on comparison-based sorting algorithms. *Arch. Formal Proofs*, 2017, 2017. URL: `https://www.isa-afp.org/entries/Comparison_Sort_Lower_Bound.shtml`.

12 Manuel Eberl. Proving divide and conquer complexities in Isabelle/HOL. *J. Autom. Reason.*, 58(4):483–508, 2017.

13 Jeff Erickson. Lecture Notes: Adversary Arguments, 2013. URL: `https://jeffe.cs.illinois.edu/teaching/algorithms/notes/13-adversary.pdf`.

14 Ruben Gamboa and John Cowles. A mechanical proof of the Cook-Levin theorem. In *International Conference on Theorem Proving in Higher Order Logics*, pages 99–116. Springer-Verlag, 2004.

15 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

16 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998.

17 Laura Kovács, Hanna Lachnitt, and Stefan Szeider. Formalizing graph trail properties in Isabelle/HOL. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2020.

18 Shiri Morshtein, Ran Ettinger, and Shmuel S. Tyszberowicz. Verifying time complexity of binary search using Dafny. In José Proença and Andrei Paskevich, editors, *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24–25th May 2021*, volume 338 of *EPTCS*, pages 68–81, 2021.

19 David Nowak. A framework for game-based security proofs. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12–15, 2007*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag, 2007.

20 Ulrich Schöpp. A formalised lower bound on undirected graph reachability. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22–27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 621–635. Springer-Verlag, 2008.

21 Alley Stoughton and Mayank Varia. Mechanizing the proof of adaptive, information-theoretic security of cryptographic protocols in the random oracle model. In *30th IEEE Computer Security Foundations Symposium*, pages 83–99, Santa Barbara, CA, USA, 2017. IEEE Computer Society. URL: `https://github.com/alleystoughton/PCR`.

22 Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. A formal proof of PAC learnability for decision stumps. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17–19, 2021*, pages 5–17. ACM, 2021.