# External Soundness in Dafny

## Background

We are in the process of building multiple implementations of the AWS Encryption SDK (ESDK) for multiple programming languages in 2020. Our implementation strategy is to implement the majority of the library in Dafny, with a minimal shim of code in each target language to wrap the Dafny implementation with a safe and relatively idiomatic API.

This is somewhat uncharted territory for Dafny: it has excelled for years at verifying properties of entirely self-contained Dafny programs, especially in an educational context, but this is one of if not the first case of releasing production software compiled from Dafny code. Although Dafny includes an {:extern} attribute that enables external code to link with Dafny in various contexts, it introduces potential unsoundness if the external code does not actually match the Dafny specification. To date, the attribute has largely been used to include trusted internal implementations, so the risk and impact of unsoundness has been somewhat minimized. Our project, however, must allow for customer code to both invoke Dafny code and implement Dafny extension points.

The impact of unsoundness becomes severe in this case. If a Dafny method declares a parameter of type Foo, for example, where Foo is a Dafny class, then Dafny will not allow passing a null value as an argument. Once this method is compiled to a target language such as C#, however, C# will allow null to be passed. This can lead to errors and undefined behaviour deep within the Dafny runtime, potentially far from the source of the error if the null value is stored and referenced at a later time. These issues will lead to a bad customer experience as it will not be clear that their code is at fault, which in turn will lead to increased operational load for our team in order to support such customers. It also undermines customer trust to tout the advantages of applying formal verification to our code base, only to ship bugs in the shim code.

Note that this problem is not unique to the ESDK, and I intend to present a large subset of this design to the Dafny community in the form of one or more RFCs, in order to motivate the Dafny changes that will be necessary to support this solutino.

## Requirements

1. The Dafny compiler should reject unsound programs involving external declarations by default as much as possible.
2. It must be possible to attach unproven assumptions to external code. The fact that these assumptions are unproven should be as explicit as possible, to aid in understanding and tracking potential technical debt.
3. We should aim to minimize the amount of manual shim code we have to write in each target language, as this directly affects the scalability of our approach.
4. The errors we provide to customers when their code violates requirements should be as clear as possible, ideally allowing them to understand the error by only referring to the target language API documentation and not the underlying Dafny source code.
5. (Nice to have) We would be happier with a solution that allows us to separate Dafny and target language source code cleanly, such that the latter can be developed, tested and built with standard tooling for each language.

This document uses a List datatype as a running example for the various requirements and options. It is a generalization of the seq<T> built-in Dafny datatype, supporting values that may exist on the heap. The initial definition below assumes it is immutable, although we will consider the implications of allowing mutation later on. It uses a 64-bit integer for length and a 8-bit integer for values, to provide common concrete types for external implementations.

const twoToThe8 := 0x1\_00  
newtype uint8 = x | 0 <= x < 0x1\_00  
  
const twoToThe64 := 0x1\_0000\_0000\_0000\_0000  
newtype uint64 = x | 0 <= x < 0x1\_0000\_0000\_0000\_0000  
  
trait List {  
 ghost const values: seq<uint8>  
 predicate Valid()  
 ensures Valid() ==> |values| < twoToThe64  
  
 function method Length(): uint64  
 requires Valid()  
 ensures Length() == |values| as uint64  
   
 method Get(i: uint64) returns (res: uint8)  
 requires Valid()  
 requires 0 <= i < Length()  
 ensures res == values[i]  
}

## Out of Scope

It has actually been difficult to exclude any issues as out-of-scope, because this is such a widely cross-cutting concern. In particular, we should consider the implications of any design decisions for target languages besides C#.

## Issues and Alternatives

The {:extern} attribute is currently quite overloaded. It can be applied to modules, traits, classes, methods and functions, and in most cases its effect is only to ensure that the names of these elements are preserved by the compiler, so that external code can reference them in some way. The details of how Dafny-generated code and manually-written external code are combined is largely left up to the specific target language. See Dafny GitHub issue [#469](https://github.com/dafny-lang/dafny/issues/469) for details.

We should first distiguish between two different use cases for linkage with external code, for the sake of consistent terminology:

1. *Exposed* methods, referring to methods that external code is able to invoke.
2. *Native* methods, referring to methods that may be implemented in external code.

In practice, it is difficult if not impossible to ensure that a method can be implemented in external code yet not invoked from external code. For example, it is natural for a Dafny code base to define a trait that is compiled to a C# interface, which can then be implemented using a C# class. There is nothing to stop other C# client code from creating this class and invoking its methods. In addition, there will be frequent use cases for methods that need to support both external invocation and native implementations anyway, such as the List example in this doc. Therefore, we propose focussing on this use case first. This is a two-way door: we can always support sound external-only or native-only methods in a future release of Dafny with looser restrictions, if this turns out to be strongly desired.

Most soundness issues can be addressed by forbidding all elements of Dafny method declarations that cannot be directly compiled to elements in the target language, as shown below. For all these constraints, we can take the approach of applying them to the ESDK code base while in parallel modifying the Dafny compiler to enforce them.

1. **Disallow types not directly supported by the target language/runtime**. This includes not only most subtypes but also newtype declarations that do not match a built-in type such as int exactly. Allowing a bounded type such as newtype MyType = x | -100 <= x < 0x8000\_0000 that compiles to a type such as int still introduces unsoundness, since external code could return -1000.
2. **Require all reference types used as in or out parameters are nullable (i.e. Foo? rather than just Foo)**. This is a corrolary of the previous statement, since many reference types in Dafny source code are implicitly non-null subtypes.
3. **Disallow almost all preconditions and postconditions.** The biggest threat to soundness is that most Dafny guarantees are checked statically. A fundamental feature of Dafny is pre- and post-conditions, expressed using requires and ensures declarations respectively. Currently, external methods may be invoked without necessarily satisfying their pre-conditions ([#461](https://github.com/dafny-lang/dafny/issues/461)) and the implementation of native methods may not necessarily satisfy their post-conditions ([#463](https://github.com/dafny-lang/dafny/issues/461)).

### External Invariants

The attentive reader will likely be surprised by the “almost” from rule #3 in the previous section. Many other programming languages ensure object invariants through a combination of access control (making fields private so that all access and mutation happens only within a bounded set of methods) and concurrency control (to ensure only one thread can ever observe an object in an invalid state at one time). This ensures that objects are valid by default. Dafny instead approaches this by validating that any operation that requires an object to be valid (which in practice is nearly all of them) provides proof that this is true, based on the context of the operation. Thus Dafny objects are assumed invalid unless proven valid.

Because of this, disallowing all preconditions on external methods is extremely limiting: it means that absolutely nothing can be assumed about any Dafny objects in the control flow of external methods. It is often not possible to dynamically verify a predicate such as Valid() since it usually refers to ghost state, and even if it was possible would often be prohibitively expensive. It is also dangerous, however, to allow external methods to assume objects are in a Valid() state without proof, for the same reasons that we cannot allow stray null values to pollute the Dafny runtime.

If we assume and/or enforce certain invariants about the linkage between Dafny and external code, however, it is possible allow very specific pre- and post-conditions on external methods. Let’s assume for the moment that Dafny objects can only be modified through Dafny methods, and not directly by external code referencing the internals of compiled code directly. This means if a predicate that reads some subset of the Dafny heap holds when control is passed to external code, it will remain true whenever Dafny code begins to execute again.

We can consider the implications while abusing Dafny notation. Since external code will be able to invoke exposed Dafny methods in any arbitrary order, the state of the Dafny heap after any one method must satisfy the requirements of all such methods. So:

forall m: ExposedMethod, m': ExposedMethod :: m'.ensures() ==> m.requires()

It must also be true that the initial state of a Dafny program must satisfy these requirements. In most cases we should be able to assume the heap is initially empty, but this depends on ensuring the Dafny runtime does not create any “built-in” Dafny objects that violate this assumption.

forall m: ExposedMethod :: initialHeapState() ==> m.requires()

Since native implementations will be able to turn around and invoke any externally-invokable method:

forall m: ExposedMethod, n: NativeMethod :: n.requires() ==> m.requires()

If we make the simplifying assumption above that ExposedMethod == NativeMethod and label both as *external methods*, we have:

forall m: ExternalMethod, m': ExternalMethod :: m'.requires() ==> m.requires() && m.ensures() ==> m.requires()

Therefore the pre-condition of every exposed method must be equivalent, and the post-condition of every such method must imply this pre-condition. We can refer to this single predicate as the *external invariant*. Although we may not know exactly what to make it, we at least know that the Dafny verifer should enforce that for every external method, its pre-condition must be equal to this invariant, and its post-condition at least as strong. That is:

exists p: bool ::  
 forall m: ExternMethod :: m.requires() == p && m.ensures() ==> p

### Defining the External Invariant

The question is now what to pick as the external invariant. Ideally this could be improved over time as Dafny’s completeness improves over time. Note that because the external invariant is a global invariant, some verification will have to be deferred to when separate Dafny packages are linked: if two packages define incompatible invariants, they cannot be used in the same Dafny program.

The options below use a hypothetical invariant {:extern} syntax for clarity: this is not strictly necessary for the required ESDK or Dafny changes, but may be desirable. The POCs can be found in the [extern-soundness](https://github.com/robin-aws/dafny/tree/extern-soundness/Test) branch of my Dafny fork.

#### Option 1: All externally-referenced objects are immutable

invariant {:extern} true

This is the option currently used on my [extern-soundness](https://github.com/robin-aws/aws-encryption-sdk-dafny/tree/extern-soundness) POC branch of the ESDK. The POC successfully applies the pattern to the Keyring trait by removing the reads this, Repr clause from Keyring`Valid().

This option by itself is not adequate, because there are mutable external datatypes we need to define, such as caching client suppliers and CMMs. It is worth mentioning, however, because it is the only option that does not require widespread additional pre- and post-conditions in order to prove the external invariant is maintained, at least without additional Dafny functionality. See Appendix A for this option applied to the List example.

#### Option 2: All Validatable objects are Valid() and independent

trait ExternallyValid {  
 ghost const Repr: set<object>  
 predicate Valid()   
 reads this, Repr   
 ensures Valid() ==> && this in Repr  
}  
  
invariant {:extern}   
 && forall v: ExternallyValid :: v.Valid()   
 && forall v: ExternallyValid, v': ExternallyValid :: v != v' ==> v.Repr !! v'.Repr

The invariant above asserts that all objects that implement a common ExternallyValid trait are Valid() at the point where an external method begins or ends. The clause requiring that all Repr sets are disjoint ensures that if one object is mutated, all other valid objects can be assumed to be still valid.

This approach is promising and seems like the simplest version that will support mutable objects. It does mean that one mutable object cannot contain another, but this seems like a desirable design principle anyway. A POC applying this to the List example is in progress. So far it seems to be necessary to add, at a minimum, ensures clauses to many unrelated methods that assert any new instances of ExternallyValid are Valid(). It is an open question whether Dafny heuristics could be improved so that this is assumed by default, just as the default modifies clause is the empty set.

class ArrayList extends List {  
 ...  
 method Add(element: uint8)   
 requires Valid()  
 ensures forall v: ExternallyValid :: fresh(v) && allocated(v) ==> v.Valid()  
 {  
 ...  
 }  
 ...  
}

#### Option 3: All Validatable objects are Valid()

trait {:termination false} ExternallyValid {  
 ghost const Repr: set<object>  
 ghost const ValidatableRepr: set<ExternallyValid>  
 predicate Valid()   
 reads this, Repr   
 ensures Valid() ==>   
 && this in Repr  
 && ValidatableRepr <= Repr  
 && (forall v :: v in ValidatableRepr ==> v.Repr <= Repr)  
 && (forall v :: v in ValidatableRepr && v != this ==> this !in v.Repr)  
  
 // Lemma that must be proved for each implementing class. Usually  
 // proves itself with an empty body, provided Valid() is defined  
 // in a compatible way.  
 twostate lemma IndependentValidity()  
 requires old(Valid())  
 requires unchanged(this, Repr - ValidatableRepr)  
 requires forall v :: v in ValidatableRepr && v != this ==> v.Valid()  
 ensures Valid()  
}  
  
invariant {:extern} forall v: ExternallyValid :: v.Valid()

This version relaxes the requirement that all externally-valid objects be independent, at the cost of having to prove separately that each implementing class can maintain validity even if objects in their Repr change from under them. It seems possible to prove such properties, but the POC is not yet complete enough to be sure.

### Other helpful Dafny features

1. **Discourage references to non-extern compiled elements**. The motivation here is to ensure external code does not interact with any part of the Dafny-generated code that has not been validated as safe to expose externally. This is critical to ensure the above assumption that the set of Dafny objects does not change outside of Dafny source code, and to ensure that external code cannot intentionally or accidentally call or implement methods that were not verified as safe for external use.

* Ideally, the compiler would attach access control to non-external elements so this is enforced by the target language compiler and/or runtime. This would prevent Dafny-generated code in one package from referencing Dafny-generated code in another, however, which is a long-term goal. Therefore, we should instead intentionally munge the identifiers generated for non-external elements to discourage external references. This will likely involve appending something like \_\_DAFNY\_INTERNAL\_\_ to these identifiers.

1. **Support traits extending other traits**. This seems to be necessary in practice, so that a trait such as ExternalList can in turn extend a generic trait like Validatable. Working around the lack of this feature is very difficult, since both types want to reference ghost state such as Repr.
2. **Support singleton objects**. This would allow statically-referencable state in Dafny programs. It has some of the same challenges around non-local invariants. It may also enable a more useful external invariant by tracking the set of externally-referencable objects as a static variable, instead of quantifying over the entire Dafny heap.

## One-Way Doors

There are two major categories of one-way doors, corresponding to the two aspects that are the most difficult to modify: the public API of the ESDK implementations we release and the semantics of the Dafny language itself.

The key priority is to expose an initial C# API that will meet customer’s needs without committing to patterns of use we do not want to support in the future. This will likely mean ensuring all non-trivial types are exposed as only interfaces, and using factory methods rather than constructors.

## Open Questions

* Can we enforce that traits are the ONLY way to specify exposed or native methods for simplicity? This would require singleton objects in other to implement exposed static methods, such as factory methods.
* Can we/should we change the rules around {:extern} by default in Dafny 3.0, or should we introduce new keywords/attributes?
* Can Dafny support for quantifying over the heap be improved?
  + Predicates are currently not allowed to do so, which means such quantifications have to happen directly in method specifications.
  + It might make sense to have some flavour of a creates clause that can be non-interfering with an external invariant by default.
* I’ve largely skipped over termination proofs here. We will have to accept some measure of unsoundness here since external code cannot be analyzed, but we also don’t want to give up and use decreases \* across the whole ESDK code base. Can we find a middle-ground that allows for partial termination proof of the Dafny code, possibly looking something like decreases Repr, \*?
* Is ANYTHING out-of-scope here? :)

# Appendix A: Lists using only immutable objects

module Collections {  
  
 const twoToThe8 := 0x1\_00  
 newtype {:nativeType "byte"} uint8 = x | 0 <= x < 0x1\_00  
 const twoToThe64 := 0x1\_0000\_0000\_0000\_0000  
 newtype {:nativeType "ulong"} uint64 = x | 0 <= x < 0x1\_0000\_0000\_0000\_0000  
  
 trait {:termination false} List {  
 predicate Valid()  
 ensures Valid() ==> |values| < twoToThe64  
  
 ghost const values: seq<uint8>  
  
 function method Length(): uint64  
 requires Valid()  
 ensures Length() == |values| as uint64  
   
 method Get(i: uint64) returns (res: uint8)  
 requires Valid()  
 requires 0 <= i < Length()  
 ensures res == values[i]  
 }  
  
 class SequenceList extends List {  
  
 const data: seq<uint8>  
  
 constructor(s: seq<uint8>)   
 requires |s| < twoToThe64  
 ensures Valid()  
 ensures Length() == |s| as uint64  
 {  
 data := s;  
 values := s;  
 }  
  
 predicate Valid()   
 ensures Valid() ==> |values| < twoToThe64  
 {  
 && data == values   
 && |values| < twoToThe64  
 }  
  
 function method Length(): uint64  
 requires Valid()  
 ensures Length() == |values| as uint64  
 {  
 |data| as uint64  
 }  
   
 method Get(i: uint64) returns (res: uint8)  
 requires Valid()  
 requires 0 <= i < Length()  
 ensures res == values[i]  
 {  
 res := data[i];  
 }  
 }  
}  
  
module {:extern "DafnyCollections"} ExternalCollections {  
  
 import opened Collections  
  
 trait {:extern "List"} ExternalList {  
 predicate Valid()   
  
 // TODO-RS: figure out how to enforce that this acts like a function  
 function method Length(): uint64   
 requires Valid()  
   
 method Get(i: uint64) returns (res: uint8)  
 requires Valid()  
 ensures Valid()  
 }  
  
 type ValidList? = l: List? | l == null || l.Valid()  
 type ValidExternalList? = l: ExternalList? | l == null || l.Valid()  
   
 /\*  
 \* Adapts a Dafny-internal List as an ExternalList  
 \*/  
 class AsExternalList extends ExternalList {  
  
 const wrapped: ValidList?  
  
 constructor(wrapped: List)   
 requires wrapped.Valid()  
 ensures Valid()  
 {  
 this.wrapped := wrapped;  
 }  
  
 predicate Valid()   
 {  
 && wrapped != null  
 && wrapped.Valid()  
 }  
  
 function method Length(): uint64  
 requires Valid()  
 {  
 wrapped.Length()  
 }  
   
 method Get(i: uint64) returns (res: uint8)  
 {  
 expect wrapped != null;  
 expect 0 <= i < wrapped.Length();  
 res := wrapped.Get(i);  
 }  
 }  
  
 /\*  
 \* Adapts an ExternalList as a Dafny-internal List  
 \*/  
 class AsList extends List {  
  
 const wrapped: ValidExternalList?  
  
 constructor(wrapped: ExternalList)   
 requires wrapped.Valid()  
 ensures Valid()  
 {  
 this.wrapped := wrapped;  
 }  
  
 predicate Valid()  
 {  
 wrapped != null && wrapped.Valid()  
 }  
  
 function method Length(): uint64  
 requires Valid()  
 ensures Valid()  
 ensures |values| < twoToThe64 && Length() == |values| as uint64  
 {  
 var result := wrapped.Length();  
  
 // TODO-RS: Ideally we would include `expect result >= 0;`, but  
 // we can't currently do that in a function, even though we really want that in  
 // the compiled version.  
 Axiom(|values| < twoToThe64);  
 Axiom(result == |values| as uint64);  
 result  
 }  
   
 method Get(i: uint64) returns (res: uint8)  
 requires Valid()  
 requires 0 <= i < Length()  
 ensures Valid()  
 ensures Length() == old(Length())  
 ensures res == values[i]  
 {  
 res := wrapped.Get(i);  
   
 Axiom(Length() == old(Length()));  
 Axiom(res == values[i]);  
 }  
 }  
  
 // TODO-RS: Replace with `expect {:axiom} ...` when supported.  
 lemma {:axiom} Axiom(p: bool) ensures p  
}  
  
// Sample method using Lists  
module Math {  
  
 import opened Collections  
   
 method Sum(list: List) returns (res: int)   
 requires list.Valid()  
 {  
 res := 0;  
 var i := 0;  
 var n := list.Length();  
 while i < n {  
 var element := list.Get(i);  
 res := res + element as int;  
 i := i + 1;  
 }  
 }  
}  
  
module {:extern "DafnyMath"} MathExtern {  
 import Math  
 import opened Collections  
 import opened ExternalCollections  
  
 // Exposed method for external client  
 method ExternalSum(list: ExternalList) returns (res: uint64)  
 requires list.Valid()  
 {  
 var asList := new AsList(list);  
 var result := Math.Sum(asList);  
 expect 0 <= result < twoToThe64;  
 res := result as uint64;  
 }  
}