Safe Haskell

David Terei[1] Simon Marlow[2] Simon Peyton Jones[2] David Mazières[1]

[1]Stanford University
davidt,⊥@scs.stanford.edu

[2]Microsoft Research
simonmar,simonpj@microsoft.com

Abstract

Though Haskell is predominantly type-safe, implementations contain a few loopholes through which code can bypass typing and module encapsulation. This paper presents *Safe Haskell*, a language extension that closes these loopholes. Safe Haskell makes it possible to confine and safely execute untrusted, possibly malicious code. By strictly enforcing types, Safe Haskell allows a variety of different policies from API sandboxing to information-flow control to be implemented easily as monads. Safe Haskell is aimed to be as unobtrusive as possible. It enforces properties that programmers tend to meet already by convention. We describe the design of Safe Haskell and an implementation (currently shipping with GHC) that infers safety for code that lies in a safe subset of the language. We use Safe Haskell to implement an online Haskell interpreter that can securely execute arbitrary untrusted code with no overhead. The use of Safe Haskell greatly simplifies this task and allows the use of a large body of existing code and tools.

Categories and Subject Descriptors D.3.3 Programming Languages [*Language Constructs and Features*]: Constraints; Modules, packages

General Terms Design, Languages, Security

Keywords Type safety, Security, Haskell

1. Introduction

One of Haskell's great strengths is how the language can be used to restrict the kinds of damage caused by coding errors. Like many languages, Haskell has a type system to enforce invariants and catch some forms of undefined behaviour, as well as a module system that allows for encapsulation and abstraction. More distinctively, Haskell segregates pure and impure code via the `IO` monad and facilitates the implementation of alternate monads (such as `ST`) that bound the possible side-effects of computations.

An interesting question is whether these features can be used to control the effects of not just buggy but outright malicious code. For instance, consider implementing a server that accepts and executes Haskell source code from untrusted network clients. Such a server should not allow one client's code to subvert another client's code or, worse yet, make arbitrary calls to the underlying operating system. Such a server is analogous to web browsers, which use language features of Java and JavaScript to confine executable content. However, because of Haskell's support for arbitrary monads, it can enable a broader range of confinement policies than most other systems.

In fact, Haskell comes tantalizingly close to providing a means for confining malicious code. Though the language allows arbitrary access to memory (with `peek` and `poke`) and to system calls (through a variety of standard libraries), these dangerous actions are in the `IO` monad. Hence, if untrusted code consists of pure code, or of computations in a monad more restrictive than `IO`, then `IO` actions will be off limits, on two conditions. First, the type system must have no exploitable loopholes. Second, abstractions, such as module encapsulation, must be strictly enforced; in particular, it must be possible to implement new monads in terms of `IO` actions while still being able to restrict the injection of arbitrary `IO` into such monads.

Unfortunately, neither of these conditions is met. Haskell 2010 [13] allows the import of foreign functions without `IO` types and provides `unsafeLocalState :: IO a → a`, both of which allow dangerous `IO` actions to masquerade as pure code (effectively lying about their types). Moreover, beyond the language specification, actual implementations of Haskell contain language extensions and special library functions, such as `unsafePerformIO` and `unsafeCoerce`, that provide many more ways to bypass abstraction and types. Hence, the first step in confining untrusted Haskell code is to identify a safe subset of the language and to have the compiler check automatically that untrusted code resides in the safe subset.

However, unsafe features exist for good reason. They can be used to implement safe abstractions using global variables, mutation, or foreign libraries, sometimes with much better performance than the pure alternative. A good example is the safe and widely-used `Data.ByteString` module, which, though referentially transparent, internally represents pure strings as packed byte arrays accessed with `peek`. Untrusted code should be able to use safe abstractions such as `ByteString` despite their unsafe internals. Hence, another important step in confining Haskell is deciding when an internally unsafe module can be trusted to provide an externally safe interface.

This paper describes Safe Haskell, an unobtrusive language extension designed to support the confining of untrusted code. Safe Haskell combines a safe subset of the Haskell language with a means for specifying when a module exceeding the safe subset can nonetheless be trusted to export a safe interface. Safe Haskell is also useful by itself because when there is no need to use unsafe features it is better to write code in Safe Haskell so that guarantees of type safety and referential transparency hold. Safe Haskell is by no means a limited version of the language—large swathes of existing code are already in the safe subset and our system allows us to automatically identify and label safe code.

Haskell'12, September 13, 2012, Copenhagen, Denmark.

Although the intuition is simple, the design of Safe Haskell is surprisingly subtle. Specifically, our contributions are these:

- We identify a clear distinction between *safety* and *security* in Haskell and define the guarantees Safe Haskell offers (Section 3.1). Safe Haskell provides enough guarantees such that a range of security policies can be implemented on top of it.
- We design a system of *trust*, which is necessary since some abstractions in Haskell are defined in terms of unsafe features and the compiler cannot guarantee their safety. Hence users must trust certain abstractions and we design a system whereby the chain of trust is tracked (Section 3.3).
- We identify the language features and extensions that must be disabled or restricted in order to provide the Safe Haskell guarantees (Section 3.4).
- We give two examples to show how Safe Haskell can be used in practice. Firstly we demonstrate a security policy that allows untrusted code access to a limited set of IO operations (Section 5.1) and secondly we show how to safely expose interactive evaluation of Haskell expressions on the web (Section 5.1.1).
- We show that large amounts of existing code are already in the safe subset (Section 6) and for the code that is not already safe we identify the main reasons why not.
- We provide an implementation of Safe Haskell in the Glasgow Haskell Compiler (GHC) that is shipping since version 7.2.

2. The problem

The presence of unsafe features in Haskell makes it unsuitable for use as a secure programming language. Secure programming as we mean it, is the ability to encode security policies and their enforcement within a programming language. Rather than rely on OS level protections which are generally coarse-grained and difficult to modify, we are interested in building secure systems solely in the Haskell language. This has the advantages of rapid prototyping, strong formal foundations and the ability to enforce very granular policies.

For example, there is extensive literature on expressing information-flow control in Haskell [9, 12, 18, 19, 25, 26]. A language-level approach to IFC makes it possible, for example, to express policies on users' data in a setting of web sites. Unfortunately, deploying these techniques in a real system has not been possible due to their reliance on properties of Haskell that do not hold (for the reasons described below).

More than just preventing Haskell for use in secure programming, the lack of strict guarantees also takes a toll on regular programming. Types provide a powerful form of documentation; a pure interface should be thread-safe, for example. That this generally holds in Haskell despite any formal guarantees is due to cultural norms in the community. Safe Haskell looks to codify these, so that any module potentially deviating from them is clearly marked.

We begin by articulating the challenges we address. We use the client/server example introduced in Section 1 as a way to make these challenges concrete. More precisely, suppose that a client is not trusted by a server and yet the server wishes to compile and run Haskell source code supplied by the client.

2.1 Unsafe language features

Haskell was originally designed to be a safe language, but the Haskell language that is in common use today is not safe. In particular, three of the properties that we regularly take for granted, *type safety*, *referential transparency* and *module encapsulation*, can be readily broken. For example:

```
f :: Int → Float → Int
f x y = x + unsafeCoerce y
```

will at least give unexpected answers because we unsafely coerce a `Float` to an `Int` and then add them together. Similarly, consider this side-effectful function that returns the system time:

```
usec :: IO Integer
usec = getPOSIXTime>>=return . truncate . (1000000 *)
```

Then this function:

```
f x = x + unsafePerformIO usec
```

is non-deterministic, perhaps giving different results on different calls. This non-determinism leads to all sorts of ills. For example, these two terms

```
(let { x = f 3 } in x + x)          (f 3 + f 3)
```

have different semantics – both are non-deterministic, but the former always returns an even number.

So the most basic requirement for Safe Haskell is that we should have a way to prevent the client from using dangerous language facilities. Moreover, one challenge is to be precise about what "safety" even means. For example, are IO-performing functions safe? We discuss all this in Section 3.1.

2.2 Unsafe language extensions

The safety issues of Haskell aren't limited to unsafe operations like `unsafePerformIO`. Some of the language extensions provided by GHC also have problems. For example, a very useful feature called "Generalised Newtype Deriving" can be used craftily to break the type system. This is a long-standing bug in GHC [8], but the fix is not straightforward.

Moreover, Generalised Newtype Deriving can be used to break the module boundary abstraction (i.e., encapsulation) of Haskell. The program below, for example, uses a newtype and module export control to create a list variation that should, after initial construction, only allow elements greater than a fixed minimum to be inserted:

```
module MinList (
        MinList, newMinList, insertMinList
    ) where

data MinList a = MinList a [a] deriving Show

newMinList n = MinList n []

insertMinList s@(MinList m xs) n
    | n > m     = MinList m (n:xs)
    | otherwise = s
```

However the invariants established by the module boundary can be broken. This is done by creating a newtype and a typeclass that contains a cast operation to convert from one type to the base type of our newtype. We define the implementation of this class for the base type and using GND, derive it for the newtype:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Main where

import MinList

class IntIso t where
    intIso :: c t → c Int

instance IntIso Int where
    intIso = id

newtype I = I Int deriving (Eq, IntIso)
...
```

What we have done is create a function we cannot create by hand. We now have an instance of the `intIso` function that will cast from `c (I Int)` to `c Int` for any type `c`. This function will remove the newtype from Int when the value is wrapped by another type. It manages this without needing to know anything about that type, including access to its constructor. We can now use this to violate the invariant of a `MinList` by defining the `Ord` typeclass for `I` in reverse of `Int`. Then we construct a `MinList (I Int)` and use our `intIso` function to convert it to a `MinList Int`.

```
...
-- we reverse the usual comparison order
instance Ord I where
    compare (I a) (I b) = compare b a

nums = [1,4,0,1,-5,2,3]

goodList :: MinList Int
goodList = foldl insertMinList
               (newMinList $ head nums)
               (tail nums)

badList :: MinList Int
badList = intIso $ foldl (λx y → insertMinList x $ I y)
                     (newMinList $ I $ head nums)
                     (tail nums)

main = do
    print goodList
    print badList
```

When running this code the output is:

```
MinList 1 [3,2,4]
MinList 1 [-5,0]
```

The use of GND has allowed us to create code that we otherwise cannot create by hand and we can abuse to create illegal values of a type.

In fairness, the issues with GND are due to bugs in the implementation of the extension itself, which we can reasonably expect to be fixed in the future. (A more careful design that fixes this bug is not yet implemented [28].) Nevertheless, the end result is as bad as `unsafePerformIO`: access to the feature currently allows the programmer to subvert the type system and language abstractions, so the feature cannot be a part of Safe Haskell.

2.3 Trust

At first one might think that, to guarantee safety, the client's code must use only safe constructs, *and* must only import modules that have the same property. But that is far too restrictive.

GHC offers such unsafe facilities precisely because they are sometimes required as the building blocks for safe, pure, deterministic abstractions; for example a memo function may be implemented using `unsafePerformIO`. The `Data.ByteString` module, first discussed in Section 1, is another good example: it presents a pure, list-like interface for bytestrings but internally relies on `unsafePerformIO` to implement this interface using mutable byte arrays. While the internals may be unsafe, the API it exposes is completely safe. Moreover, lazy evaluation itself depends on the benign side effect of updating a thunk and it is no less dangerous because it is implemented in the compiler and runtime system.

So in the end it is all about *trust*. We will take for granted that the server trusts the compiler and its runtime system – compiler verification is a different issue. But the server most likely also trusts some libraries, including ones such as the low-level `base` package that come with the compiler, even though these libraries may use unsafe features.

In short: *it is OK for the client's code to import modules that the server trusts.* Notice that trust relates to the person doing the compiling (the server in this case) not the author of the code. A different server might have a different opinion about which libraries should be trusted.

3. Design

Safe Haskell tackles the problem of unsafety by defining a subset of the language that *does* provide properties like referential transparency, type safety and module encapsulation, while encompassing as much existing code as possible.

The Safe Haskell design involves four related components:

- A set of guarantees that Safe Haskell code provides when compiling code in the safe language (Section 3.1),
- A categorization of Haskell modules according to what safety guarantees can be provided (Section 3.2),
- A system of *trust*, that allows safe abstractions to be implemented using unsafe features (Section 3.3),
- A set of restrictions on language features and extensions that constitutes the safe subset of the language (Section 3.4)

3.1 Safe language guarantees

The purpose of the safe language is *to allow the programmer to trust the types*. In the safe language, subject to trust, a function that claims to be a pure function is guaranteed to indeed be pure. Module export lists can also be thought of as type signatures—as they are in the ML community—and so the above simple definition applies to them too. These guarantees are dependent on the trusted base that a user chooses to enable (Section 3.3). Subject to this, Safe Haskell provides the following guarantees for code compiled in the safe language:

- *Type safety.* In Milner's famous phrase, well-typed programs do not go wrong.
- *Referential transparency.* Functions in the safe language must be deterministic. Evaluating them should not cause any side effects, but may result in non-termination or an exception.
- *Module encapsulation.* Haskell provides an effective module system that is used to control access to functions and data types. In the safe language these module boundaries are strictly enforced; a user of a module that contains an abstract data type is only able to access or create values through the functions exported.
- *Modular reasoning.* Adding a new import to a module should not change the meaning of existing code that doesn't directly depend on the imported module.
- *Semantic consistency.* Any valid Haskell expression that compiles both in the safe language and in the full language must have the same meaning.

The reader may find it surprising that Safe Haskell does not restrict the use of IO. For example, it is perfectly possible to write a program in Safe Haskell that deletes all the user's files. This is a deliberate design decision: a property like "does not delete all the users files" is a *security property* and in general the security policy that a given application requires will depend on that application. Instead, Safe Haskell provides sufficient guarantees that a range of security policies can be implemented on top of it. We will discuss such security mechanisms in Section 5.1.

3.2 Identifying Safe Haskell code

The universe of Haskell code consists of code that is in the safe language and code that is not. How should this distinction be indicated by the programmer? The key design choice we made here is to identify safety at the granularity of a Haskell *module*; this makes it easy both for the programmer and the compiler to see the boundary between safe and unsafe code.

Every module belongs to one of the following three classes:

- `Safe`, indicating that the module is written in the safe language.
- `Trustworthy`, indicating that the module is not written in the safe language but nevertheless its author claims that clients can use it safely (we discuss this in more detail in Section 3.3).
- `Unsafe`, for all other modules.

The programmer may explicitly specify which class their module belongs to by specifying `Safe`, `Trustworthy`, or `Unsafe` in the `LANGUAGE` pragma at the top of the module. For example:

```
{-# LANGUAGE Safe #-}
```

Equivalently, the server compiling an untrusted module can pass the `-XSafe` flag to GHC.

In the absence of a `Safe`, `Trustworthy`, or `Unsafe` indication, the compiler will automatically infer either `Safe` or `Unsafe` at compile-time according to whether the code of the module lies within the safe language or not.

3.3 Trust

Suppose the compiler is compiling this module:

```
{-# LANGUAGE Safe #-}
module M ( f ) where

import A ( g )
import B ( y )

f x = g x + y
```

Because M uses Safe Haskell, the compiler checks two things:

- It checks that M's code, here the definition of `f`, does not contain any unsafe language features.
- It checks that any modules M imports, here `A` and `B`, are *trusted* modules. We will say what we mean by "trusted" shortly, but the intent is that trusted code respects the Safe Haskell guarantees enumerated in Section 3.1.

What does it mean for a module to be "trusted"? One obvious reason for the compiler to trust A is that A is declared `Safe` and was successfully compiled. However, as discussed in Section 2.3, we need a way for the code to depend on unsafe code that is trusted by the person who is going to run the compiled code.

In Haskell, modules are typically grouped together into packages, which are the unit of distribution and also usually the unit of authorship. Hence it makes sense to track trust at the level of a package. So the decision about whether "the user U trusts module M in package P" is divided into two parts, one for the module and one for the package.

These considerations lead us to the following definitions. A module M in a package P is *trusted* by a user U if either of these two conditions hold:

1. The compiler can check that M is indeed worthy of trust. Specifically, both of these hold:
   - The module was declared to be `Safe`.
   - All of M's direct imports are trusted by U.
2. The user U trusts M. Specifically, all of these hold:
   - Package P is trusted by U; we describe in Section 4.1 how U expresses this choice.
   - The module was declared by its author to be `Trustworthy`.
   - All of M's direct *safe imports* are trusted by U; we describe safe imports in Section 3.3.1.

Even a trusted author may design a package that exposes both `Safe` and `Unsafe` modules, so the second test is that the (trusted) author declared that this particular module M is indeed a `Trustworthy` one, using a pragma:

```
{-# LANGUAGE Trustworthy #-}
```

A module that is declared to be `Trustworthy` is claimed by the author to expose a safe interface, even though its implementation might make use of unsafe features. There are no restrictions placed on the language features used by, or the modules imported by, a `Trustworthy` module (although the author can explicitly request that certain imports must be trusted, see Section 3.3.1).

3.3.1 Safe imports

Suppose there is a `Trustworthy` module M in a package P that is trusted by the server and accessible to untrusted client code. Moreover, M depends on module N from another package Q. Now we can see two possible situations:

- P's author trusts Q and is willing to take full responsibility. In that case, M can just import N as usual.
- P's author is not willing to take responsibility for Q. What P's author wants to say is "if the server trusts Q, then my modules are fine". For that case we provide a small extension to Haskell's import syntax:

  ```
  import safe N
  ```

The sole effect of the `safe` keyword can be seen in the definition of a trusted module (Section 3.3): the server only trusts M if it trusts M's author, M's author claims M is `Trustworthy` *and* the server trusts all M's `safe` imports.

When compiling with the `Safe` pragma, it is not necessary to annotate imports with `safe`, since all imports are required to refer to trusted modules. While the same could have been done for `Trustworthy` modules and require some imports be marked as `unsafe`, in reverse of the current design, this would have introduced a larger, non-backward-compatible change to the language.

3.3.2 Example

```
Package Wuggle:
    {-# LANGUAGE Safe #-}
    module Buggle where
        import Prelude
        f x = ...

Package P:
    {-# LANGUAGE Trustworthy #-}
    module M where
        import System.IO.Unsafe
        import safe Buggle
        ...
```

Suppose a user U decides to trust package P. Then does U trust module M? To decide, GHC must check M's imports.

- M imports `System.IO.Unsafe`, but M was declared to be `Trustworthy`, so P's author takes responsibility for that import.
- U trusts P's author, so U trusts M to only expose an interface that is safe and consistent with respect to the Safe Haskell guarantees.

- *M* also has a safe import of `Buggle`, so for this import *P*'s author takes no responsibility for the safety, so GHC must check whether `Buggle` is trusted by *U*. `Buggle` itself has a `Safe` pragma so the module is machine-checked to be OK, but again under the assumption that all of `Buggle`'s imports are trusted by *U*. We can probably assume that `Prelude` is a trusted module and the package it resides in is trusted, thus `Buggle` is considered trusted.

Notice that *U* didn't need to trust package `Wuggle`; the machine checking is enough. *U* only needs to trust packages that contain `Trustworthy` modules.

3.4 Restricted language features

In the following sections we discuss the language features that are either restricted or completely disallowed by the safe language.

3.4.1 Unsafe functions

A module in Safe Haskell cannot have access to `unsafePerformIO`, because that would allow it to violate *referential transparency* and break *type safety*. The same applies to a number of GHC's primitive operations and a handful of other operations in the libraries that come with GHC. The modules that expose these unsafe functions are regarded by GHC as `Unsafe`, preventing them from being imported by `Safe` code.

3.4.2 Disallowed extensions

While Haskell 2010 with the exception of non-IO FFI imports and `Foreign.Marshal.unsafeLocalState` is a safe language, most Haskell compilers implement a variety of widely used extensions to the language, a few of which are unsafe. These extensions violate the guarantees of Safe Haskell and so are disabled when compiling with the safe language. These extensions are:

- *Template Haskell* – Allows access to symbols in modules regardless of the export list they define. This breaks *module encapsulation*.
- *Generalised Newtype Deriving* – Allows the deriving of code that otherwise cannot be written by hand. This can be used to create functions that operate on values contained within abstract types as we demonstrated in Section 2.2, breaking *type safety* and *module encapsulation*.
- *RULES* – Allows redefining the meaning of existing code in the system. That is, importing a module that uses RULES may change the behaviour of existing code in the importing module. This breaks the *modular reasoning* guarantee.

3.4.3 Restricted extensions

While some Haskell language extensions are unsafe and need to be disabled entirely, a few extensions can instead be restricted in their functionality to ensure they are compatible with the guarantees offered by Safe Haskell. These restrictions do not change the behaviour of these extensions, they simply limit it. Should `Safe` code exceed these limits, a compile-time error occurs. The restricted extensions are:

- *Foreign function interface* – the FFI allows importing functions with non-IO type signatures, which is useful for importing pure foreign functions such as `sin`. This essentially amounts to use of `unsafePerformIO`, the programmer is asserting that the function is pure and the compiler cannot check the assertion. Hence in Safe Haskell, all FFI imports must have IO types.
- *Deriving Data.Typeable* – Data.Typeable is part of the widely used Scrap Your Boilerplate [11] (SYB) generic programming techniques. The `Typeable` typeclass gives to each type that is a member of the class a unique, comparable representation, that is used to implement type-safe casts on top of an unsafe cast primitive. The original SYB paper envisioned that safety could be guaranteed by having the compiler derive the `Typeable` instance for a type. While GHC support this, it also allows the programmer to define their own instances of `Typeable`. Hand-crafted `Typeable` instances can be used to cast between arbitrary types, undermining *type safety* and *module encapsulation*. Safe Haskell prevents this by only allowing automatically derived instances of `Typeable`.
- *Overlapping Instances* – Allows redefining the meaning of existing code in the system, thereby breaking the *modular reasoning* guarantee. The safe language restricts this by only allowing instances defined in a module M to overlap with instances also defined in module M. Should any of the instances in M overlap with instances defined outside of M, a compile-time error will occur.

4. Implementation details

In addition to the core design of Safe Haskell, there were several decisions made that fall on the implementation side but form an important part of the functionality of Safe Haskell. Several of these decisions were made with the benefit of hindsight as Safe Haskell was first implemented and released in GHC 7.2, but based on feedback from users and our own experience we modified the implementation, arriving at the one detailed in this paper and released in GHC 7.4.

4.1 Package trust and `-fpackage-trust`

In the first implementation of Safe Haskell we discovered a significant drawback: a library author would add a `Safe` pragma to their modules and upload the library to Hackage for others to use. When users came to compile the package the compilation would fail because one or more packages were not trusted on the user's machine. We believe it is important that library authors are able to add `Safe` pragmas to their code without the risk that this will cause compilation failures for their users for reasons that are outside the library authors control.

This lead to an important realisation: the checking of package trust can be deferred until required by a user who actually cares about the chain of trust. For a user compiling a package they downloaded from the Internet and who is otherwise not using Safe Haskell, it is important that the compilation not fail for a spurious reason. Hence in the implementation of Safe Haskell we made the checking of package trust conditional on the `-fpackage-trust` flag, which is off by default. When the flag is off, every package is effectively considered to be trusted.

It is important to understand that this policy does not weaken the guarantees provided by Safe Haskell, it only defers the check that `Trustworthy` modules reside in trusted packages. Consider a chain of modules where `A` imports `B` which imports `C` and both `A` and `B` are `Safe` and `C` is `Trustworthy`. When compiling `B` it is irrelevant whether the package containing `C` is currently trusted, because when we eventually compile `A`, we will make that check again, because the definition of a trusted module is recursive. So the observation is that we can safely assume all packages to be trusted when compiling intermediate modules like `B`, but a user who cares about the chain of trust can enable `-fpackage-trust` when compiling the top-level module `A` and have no loss of safety.

Finally, this feature does not mean that compilation of a package will never fail due to Safe Haskell if the user of the package has chosen not to enable Safe Haskell. When the author of a package *P* uses a `Safe` pragma in their code they are still placing a requirement that the modules their code depends on are consid-

ered `Safe` or `Trustworthy`. If a new version of a package that *P* depends on is released with a change to which modules are considered `Safe` or `Trustworthy`, this may now cause *P* to fail to compile. Safe Haskell is part of the versioning of a package. The `-fpackage-trust` flag simply removes package trust from the equation for package authors as this is local to a users machine and not within their control.

4.1.1 Package trust options

Packages in Haskell have a variety of metadata attached to them that is stored in a global database known as the package database. This metadata consists of static information but also properties that affect the behaviour of GHC, such as which package to choose when there is a choice during compilation. These properties can be set permanently against the package database or for a single run of GHC.

As explained in Section 3.3, Safe Haskell gives packages a new Boolean property, that of trust. Several new options are available at the GHC command-line to specify the trust of packages:

- `-trust P` – Exposes package *P* if it was hidden and considers it a trusted package regardless of the package database.
- `-distrust P` – Exposes package *P* if it was hidden and considers it an untrusted package regardless of the package database.
- `-distrust-all-packages` – Considers all packages distrusted unless they are explicitly set to be trusted by subsequent command-line options.

4.2 Safety inference

As mentioned in Section 3.2, the safety of a module can be inferred during regular compilation. That is, for every compilation that GHC performs, it tracks what features of Haskell are being used and what modules are being imported to determine if the module can be regarded as `Safe`. The precise condition for a module to be inferred as `Safe`, is that compiling that same module with the `Safe` pragma and *without* the `-fpackage-trust` flag would succeed. In other words, modules are inferred as `Safe` or `Unsafe` under the assumption that all `Trustworthy` modules reside in trusted packages.

4.2.1 `Unsafe` modules

Safe inference introduces the possibility of conflicting uses of Safe Haskell, which, on one hand tracks and guarantees simple type safety and on the other can be the basis of a particular security policy. The problem arises when a module `M` is perfectly type-safe but exports private symbols that must not be imported by untrusted code. For instance, a useful idiom is to have a `privilegedLiftIO` function that allows execution of arbitrary `IO` actions from a more restrictive monad. Such a function might be type-safe, but must not be imported by untrusted code—it is only for use by privileged code implementing the restrictive monad's API.

Since GHC will automatically infer a module as `Safe` if it resides in the safe subset, it may be necessary to explicitly label a module as `Unsafe` if the module's interface would allow untrusted code to break security properties. Another way to achieve this effect would be to import a known-unsafe module such as `System.IO.Unsafe`, but explicitly labelling the module as `Unsafe` is clearer. The `Unsafe` pragma also enables the use of the `safe` keyword in imports and hence is useful for `Main` modules importing untrusted code.

4.3 Haskell tools

In addition to the core design of Safe Haskell we have spent some effort on the surrounding infrastructure of the GHC compiler, namely GHCi and Haddock.

For GHCi we added full support for Safe Haskell; invoking GHCi with the `-XSafe` flag functions as one would expect. Imports during the session are checked appropriately for trust. We also added an `:issafe` command to GHCi that can be used to check the Safe Haskell status of a module. For example:

```
Prelude> :issafe Data.ByteString
Trust type is ( Module: Trustworthy
              , Package: untrusted )
Package Trust: Off
Data.ByteString is trusted!
```

We also added support for Safe Haskell to the Haddock documentation tool. Haddock now automatically includes the Safe Haskell categorization of a module in its generated documentation. If a module is inferred to be `Safe` by GHC, then the module's documentation will indicate so.

5. Use cases

Safe Haskell has been designed with a number of uses cases in mind:

- Enabling (given a set of trusted components) secure systems that host untrusted Haskell code to be built and relied upon (Section 5.1).
- Encouraging language designers to carefully consider the safety implications of new language extensions. Those features that allow the Safe Haskell properties to be broken have serious implications; again, Safe Haskell shines a light on this (Section 5.2).
- Providing an additional tool to encourage writing in the safe subset of Haskell, something we believe should be considered best practice. Leaving the safe language should be a conscious decision and Safe Haskell provides a framework for implementing this policy (Section 5.3).

5.1 Secure programming

The primary motivation for the work on Safe Haskell has been to harden the language to a point at which it can be used for the design and implementation of secure systems; that is, systems designed to sandbox third party untrusted code. Haskell's powerful type system, in particular its separation of pure and impure code and the ability of typeclasses to encode monads, makes it well-suited for such a task. These allow certain security systems to be built in Haskell that would otherwise require compiler or OS support in other languages.

An example used to drive Safe Haskell was that of implementing a restricted `IO` monad. A restricted `IO` monad, or `RIO`, is the next simplest example of a viable sandbox mechanism in Haskell after a pure function. The idea of `RIO` is that third party untrusted Haskell code can be run safely by requiring it to provide a top level interface (the plugin interface) that resides in a new monad that is isomorphic to `IO` but, by creating a distinct type, controls precisely what `IO` actions are available to the untrusted code.

An example encoding of this idea can be seen below:

```
{-# LANGUAGE Trustworthy #-}
module PluginAPI (
        RIO(), runRIO, rioReadFile, rioWriteFile
    ) where

-- Notice that symbol UnsafeRIO is not exported
-- from this module!
newtype RIO a = UnsafeRIO { runRIO :: IO a }

instance Monad RIO where
    return = UnsafeRIO . return
    (UnsafeRIO m) >>= k = UnsafeRIO $ m >>= runRIO . k
```

```
-- Returns True iff access is allowed to file name
pathOK :: FilePath → IO Bool
pathOK file = {- Implement some security policy -}

rioReadFile :: FilePath → RIO String
rioReadFile file = UnsafeRIO $ do
  ok ← pathOK file
  if ok then readFile file else return ""

rioWriteFile :: FilePath → String → RIO ()
rioWriteFile file contents = UnsafeRIO $ do
  ok ← pathOK file
  if ok then writeFile file contents else return ()
```

An untrusted module can be compiled with the `Safe` pragma and run with safety as long as only functions residing in the `RIO` monad are called. In this situation a guarantee is provided that the only `IO` actions that can be executed by the untrusted code are `rioReadFile` and `rioWriteFile` both of which are protected by the `pathOK` function. Essentially we are restricting the plugin to a sandboxed filesystem, all within the Haskell language.

Now it is worthwhile walking through ways in which the above code would fail without Safe Haskell. A malicious module has at least the following attack options available:

- `unsafePerformIO` could be used to pretend any `IO` action is a pure function and thus execute it in the `RIO` monad.
- The FFI could be used to import an impure foreign function with a pure type signature, allowing it to be used in the `RIO` monad.
- A hand-crafted instance of `Data.Typeable` could be defined for `RIO` in the malicious module such that its type value is equivalent to `IO`. The cast operation could then be used to coerce any `IO` action to a `RIO` action.
- Template Haskell can be used in to gain access to the `RIO constructor`, allowing any `IO` action to be constructed as a `RIO` action.

Compiling with Safe Haskell is designed to effectively prevent all such attacks and allows the `RIO` sandbox to work as expected.

A more powerful sandbox mechanism than the simple `RIO` technique that can be encoded in Haskell is that of information-flow control, or IFC. The LIO [19] library for Haskell offers a powerful IFC mechanism for using as a basis for building secure systems. Indeed building on top of Safe Haskell and LIO several authors of this paper are involved in designing and implementing a web framework that supports building websites that include untrusted third party plugins while still guaranteeing the site's users privacy policies are respected.

5.1.1 GHCi Online

As an example of the use of Safe Haskell we designed and implemented a version of the interactive Haskell environment, GHCi, that can be run as a web site for use by unknown, untrusted users. That is, it offers an online service for executing untrusted Haskell code. This is inspired by such existing services of this nature like LambdaBot [21] and `TryHaskell.org` [3].

GHCi Online [1] builds on the `RIO` technique outlined Section 5.1. This differs from LambdaBot and TryHaskell which use syntax level filtering and take a fairly heavy-handed approach of disabling all `IO`. As we had already modified GHCi to support Safe Haskell as part of the project, a large portion of the implementation work for GHCi Online was already done. We did however add support to GHCi to execute within the bounds of a user specified monad rather than the default `IO` monad.

To support a *read-eval-print* loop, GHCi's design includes a source level transformation that lifts any user typed expression into the `IO` monad for execution. Aspects like binding variables and printing out results are also handled as part of this transformation. The rules for it are:

```
User expression
⇒
The IO [Any] that is run
--------------------------

1. let (x,y,...) = expr
   ⇒
   let (x,y,...) = expr in return [coerce Any x,
                                   coerce Any y,
                                   ...]

2. (x,y,...) ← expr
   ⇒
   expr >>=λ(x,y,...) → return [coerce Any x,
                                coerce Any y,
                                ...]

3. expr -- (of IO type)
   ⇒
   expr >>=λit → return [coerce Any it]

4. expr -- (of non-IO type, result showable)
   ⇒
   let it = expr
   in print it ≫ return [coerce Any it]

5. expr (of non-IO type, result not showable)
   ⇒
   error
```

These rules deal with binding values and coercing them to GHC's `Any` type, which a type that can hold a dynamically typed value and is used by GHC for a variety of purposes, including implementing the `Data.Typeable.Dynamic` data type. We can see in rules 1 and 2 how interactively binding variables is done. Rule 3 demonstrates the simple transformation done to `IO` expressions, simply capturing the result in a variable named `it`, which is a convenience of GHCi. Rule 4 shows how expressions of non-`IO` type that are printable are executed, while rule 5 shows how expressions that aren't an `IO` type or of the `Show` typeclass produce an error.

We generalised this transformation so that GHCi was able to lift and execute code not just in the `IO` monad, but into any monad that was isomorphic to `IO`. We captured the requirements for this in a typeclass and modified the source transformation to support executing expressions within any monad that is an instance of the typeclass. The definition of this typeclass appears below:

```
module GHC.GHCi (
        GHCiSandboxIO(..), NoIO()
    ) where

import GHC.Base (IO(), Monad, (>>=), return, id, (.))

-- | A monad that can execute GHCi statements by
-- lifting them out of m into the IO monad.
class (Monad m) ⇒ GHCiSandboxIO m where
    ghciStepIO :: m a → IO a

instance GHCiSandboxIO IO where
    ghciStepIO = id

-- | A monad that doesn't allow any IO.
newtype NoIO a = NoIO { noio :: IO a }
```

[1] GHCi Online can be found at `http://ghc.io`

```
instance GHCiSandboxIO NoIO where
    ghciStepIO = noio

instance Monad NoIO where
    return a  = NoIO (return a)
    (>>=) k f = NoIO (noio k>>=noio . f)
```

The generalisation of GHCi and the definition of the typeclass interface gave us an easy and flexible way to sandbox GHCi itself. To do so, we simply implement a `RIO` style monad and make it an instance of the `GHCiSandboxIO` typeclass. Currently we have not implemented a `RIO` style monad for GHCi Online but simply restricted `IO` completely by using the `NoIO` type defined above. For future work we would like to use a `RIO` style monad that implemented `IO` functions like `getChar` as callbacks between the browser and server-side of GHCi Online.

Compared to LambdaBot and TryHaskell, GHCi Online represents a principled and flexible approach to safely executing untrusted Haskell code over the Internet. The security policy consists simply of the 6 lines of code that is needed to implement the `NoIO` monad. Implementing a more powerful sandbox would be a simple task. By comparison LambdaBot and TryHaskell both rely strongly on a white list of modules that can be imported to assure type safety and only support executing pure functions. Changing the policy and code is far from trivial for these systems.

5.2 Quality mark

A more ambitious and less concrete use case for Safe Haskell is for it to act as a quality bar for the Haskell language. Many Haskell programmers may be surprised at the number and subtle ways in which certain language extensions can be used to circumvent the guarantees Haskell is usually thought to provide. While `unsafePerformIO` is not one of them, the ability of Generalised Newtype Deriving or Template Haskell to break module boundaries likely is.

The Haskell language has always walked a line between acting as a research vehicle for programming language design, while increasingly being used by industry for commercial purposes. Safe Haskell has the potential to help define part of that line, by marking which extensions at least conform to the guarantees Haskell provides. It offers a focal point for discussing some of the more controversial features of Haskell, helping to bring some of these issues to light. New extensions to the language and new packages will now face the question of their Safe Haskell status, hopefully encouraging a stronger design.

5.3 Good style

As we discussed in Section 2, the use of unsafe functions and language extensions can lead to programs that have undesirable behaviours (e.g., crash). As such it is considered good practice in the Haskell community to use unsafe features only if no appropriate safe alternative exists. Safe Haskell and the `Safe` pragma can be used to codify this practice. In this sense we can think of Safe Haskell as another kind of "warning." It is already common for developers to turn on GHC's `-Wall` flag that emits warnings for stylistic aspects that are generally disapproved of, often because they can hide bugs (e.g., unused identifiers). In the same way that it is considered good to keep one's code warning-clean, we expect that using `Safe` will become another aspect of code cleanliness that will find its place amongst the accepted best practices.

There may be some initial friction as developers adjust to using `Safe` and `Trustworthy`; there will no doubt be many libraries that need small changes to work with `Safe`. However we expect that over time this should change as Safe Haskell sees broader adoption and hopefully some of the restrictions can be dropped through further work on their design or restrictions on their functionality under Safe Haskell. This would further encourage a trend towards using `Safe` by default.

	Modules	% of `base`
Safe	23	25.56%
Trustworthy	59	65.56%
Split	5	5.56%
Unsafe	3	3.33%

Table 1. Results of using Safe Haskell in GHC `base` package.

6. Evaluation

To evaluate the ease of using Safe Haskell, particularly when porting existing code to the safe language, we looked at three different metrics:

- First, we manually added Safe Haskell pragmas to the `base` package and several other packages distributed with GHC (Section 6.1).
- Second, we determined what fraction of the Haskell world (as determined by Hackage) is inferred as `Safe` (Section 6.2.1).
- And third, we determined what fraction of the Haskell world compiles with the restrictions to language extensions the `Safe` pragma applies but without any restrictions on imports (Section 6.2.2).

6.1 Making `base` safe

We modified the `base` package distributed with GHC, that includes the Haskell standard prelude, to use Safe Haskell.

Each module was either inferred as `Safe` or `Unsafe`, marked as `Trustworthy`, or split into two new modules such that the safe symbols resided in one module and the unsafe symbols in another. This split was done such that a module `M` produced a new module `M.Safe` containing the safe symbols and a new module `M.Unsafe` containing the unsafe symbols. `M` itself was left unchanged to preserve backward compatibility and is inferred as `Unsafe`, however future versions of the `base` package will move towards `M` only exporting safe symbols.

The results of this process for the `base` package can be seen in Table 1. As is expected for `base`, which deals with many of the lowest levels of the Haskell language, a large proportion, 66%, of the modules needed to be marked `Trustworthy`. For the remaining, 26% of the modules were inferred as `Safe`, 6% of the modules were split, 3%, or 3 out of 89 modules, were inferred as `Unsafe`. The split modules include, for example, `Control.Monad.ST` as it contains the unsafe function `unsafeIOToST`. While the unsafe modules were `Data.Debug`, `System.IO.Unsafe` and `Unsafe.Coerce`, since they only contain unsafe symbols.

6.2 Compiling Hackage

To see the effect of Safe Haskell on the wider Haskell universe we tried compiling every package (totalling over 14,000 modules) on the Hackage code hosting site. For one run we simply recorded which modules were inferred to be `Safe` and which were inferred to be `Unsafe`. For a second run we compiled all packages using a modified version of the safe language in which all the language extension restrictions applied but import control was turned off. To perform these tests we improved upon an existing tool called Hackager [24] that automates the process of compiling all of Hackage.

6.2.1 Inferring `Safe`

The results for the first run of Hackager can be seen in Table 2. Around 27% of modules on Hackage are inferred as `Safe` while

	Modules	% of Hackage
Safe Inferred	3,985	27.21%
Unsafe Inferred	10,660	72.79%

Table 2. Results of inferring Safe Haskell status for all modules on Hackage.

	Packages	% of `Hackage`
Buildable	1,278	82.66%
Build failed	268	17.34%

Table 3. Results of compiling all packages on Hackage with a modified `Safe` pragma (no import restrictions).

72% are inferred as `Unsafe`. While we think this is already an encouraging number, we expect the number of modules inferred as `Safe` to grow due to a number of factors.

Firstly, as mentioned in Section 6.1, some of the modules in the `base` package of GHC needed to be split into a safe and unsafe module. To preserve backward compatibility we left the original module unchanged and hence inferred as `Unsafe`. Instead of this, future versions of the `base` package will remove the unsafe symbols from the primary module so that it can be regarded as `Safe` or `Trustworthy`. Since few users of the `base` package rely on unsafe symbols, switching the primary module to the safe interface should allow more of Hackage to be inferred as `Safe`.

Secondly, Safe Haskell is a very recent addition to GHC and it is reasonable to expect that right now represents the lowest point in its adoption by the Haskell community. As time passes, we believe that more package maintainers will either use the `Trustworthy` pragma when appropriate or, as is our hope, refactor their code so that most of the modules can be inferred as `Safe`.

6.2.2 Evaluating the `Safe` pragma

To evaluate the impact of using the safe language we performed a second run of Hackager in which we compiled each package using a modified version of the `Safe` pragma. This modified version kept the language extension restrictions but dropped the safe import requirement. That is, using Template Haskell would cause a compile-time failure but importing `System.IO.Unsafe` would not.

The results of this test can be seen in Table 3. We present these results at the package granularity, showing the number of whole packages that successfully compiled or failed to compile. Furthermore, we only compiled the package being tested with the modified `Safe` pragma; the dependencies of each package were compiled without any use of Safe Haskell. Thus this test is more an indication of the degree to which restricted language extensions are used, than an indication of what packages truly compile in a `-XSafe` restricted world. The results however, are very encouraging, with 83% of packages compiling successfully.

Furthermore, we broke down the reason for packages failing to build with the results presented in Table 4 [2]. As can be seen from the table, the use of Generalised Newtype Deriving accounts for around half of the packages that failed to build. Since we believe this restriction can be lifted in the future it offers further evidence that Safe Haskell can see broad adoption by the community with little friction.

[2] Some packages failed to build for more than one reason, hence the larger total than packages that failed to build

	Modules	% of failure
Generalised Newtype Deriving	146	54.48%
Template Haskell	84	31.34%
Hand written Typeable instances	33	12.31%
Non-IO FFI imports	33	12.31%

Table 4. Count of packages that failed to build with a modified `Safe` pragma (no import restrictions) summed by the language extension that caused the failure.

7. Discussion

During the work on Safe Haskell, a number of alternative designs were considered before we arrived at the system presented in this paper. We discuss them here.

One choice was the granularity at which to consider safety. Instead of marking safety at the module level, the safety of individual symbols could be tracked. A proposed syntax for this was:

```
{-# LANGUAGE Trustworthy #-}
module M where (
        {-# SAFE -#}
        a,b,c,d

        {-# UNSAFE -#}
        e,f,g
    )
...
```

This design would allow for safe and unsafe symbols to co-exist within the same `Trustworthy` module instead of requiring they be split into two different modules as the current design does. A safe import of the module would only have access to the safe symbols while, a regular import would have access to both the safe and unsafe symbols.

This design has a number of advantages, the first being that it appears easier for existing code to adopt, because no changes to the module layout of a package are needed. Since the current design of Safe Haskell may require splitting module *M* into three modules, an implementation, a `Trustworthy` module and an `Unsafe` module this is an important advantage. The advantage also benefits users of *M*, who would not need to change any of their imports. Secondly, under the alternative design we can generate better error messages. A call to the unsafe function `e` when *M* is imported with a safe import would produce a compile-time error message that the use of an unsafe function is not allowed. In the current design of Safe Haskell the error message instead reports that `e` is not in scope. Despite these advantages, we did not take this approach as it is a more invasive change to the Haskell language than the current design. Using modules as the level of granularity doesn't introduce any new name-space concepts in the language and builds on an existing idiom in the Haskell community to place unsafe functionality in its own module. It also imposes a larger overhead on using unsafe symbols and hence may help to dissuade users from doing so.

A second major decision in the design of Safe Haskell is to have a single notion of "trustworthy", rather than generalising to arbitrary types of trustworthiness. The current design of Safe Haskell places a Haskell module in one of three sets: `Safe`, `Trustworthy` or `Unsafe`. The `Trustworthy` set has always been more questionable than the `Safe` or `Unsafe` sets with the issues largely revolving around the idea that one person may have a different definition of "trustworthy" than another. It may also be desirable to specify different levels of trustworthiness, some with stronger or weaker guarantees than others. Along these lines, we explored a design in which we generalised the notion of categorizing modules by having arbitrary, user defined sets. One proposed syntax for this was:

```
{-# SET Trust_T1 #-}
module A where ...

{-# SET Trust_T2 #-}
module B where ...
```

In this design, the `Safe` set would be a special set functioning as it does in the current design but the `Trustworthy` and `Unsafe` sets would simply become convention. Other types of trustworthiness could also be defined by individual package maintainers. This design had some appeal in the increased flexibility and power it offered users. We ultimately decided against it though for two reasons. Firstly, the design of Safe Haskell is subtle and at times complicated; this alternative adds greatly to the complexity. Secondly, having a universally shared definition of a `Trustworthy` module is extremely beneficial. The increased flexibility doesn't just increase the complexity in using Safe Haskell but also in validating what guarantees it provides and assuring that the system behaves as expected.

Another design decision of interest is that of import control in Safe Haskell. The current design offers a weak version of a white-listing import control mechanism, in the form of the `-fpackage-trust` flag. By marking packages as trusted or untrusted, the importing of `Trustworthy` modules can be controlled. An alternative would be to extend this mechanism so that it is possible to compile a module with a guarantee that it could only import modules on a list defined by the server invoking GHC. If we also allowed for all FFI imports to be disabled then some stronger static guarantees can be provided than the current design of Safe Haskell. An example of this could be the assurance that `IO` actions in a safe module were only constructed from existing `IO` actions in modules it can import. Rather than rely on a restricted `IO`-style sandbox, the `IO` monad could be used directly in untrusted code. We decided against this design, because we believe it would add extra complexity to Safe Haskell with little benefit. In the end the advantages of Safe Haskell for building secure systems is not a simple import control mechanism, but rather the advanced type system of Haskell that can encode ideas such as reduced `IO` monads and information-flow control.

The last design decision we will discuss is that of requiring explicit use of the safe language through `-XSafe` or the `Safe` pragma, rather than making it the default. Instead of allowing the use of unsafe language features at any point in modules not specifying a Safe Haskell pragma, we could require modules explicitly enable access to unsafe features through the `Unsafe` pragma. While appealing in requiring users opt-out of safety as opposed to having to opt-in, such a design would break a large amount of Haskell code, making it impractical to deploy.

8. Limitations

While Safe Haskell provides a powerful and comprehensive mechanism for implementing secure systems, the Haskell language, there are a number of restrictions it implies and limitations on its ability to achieve this without supporting tools.

The first limitation of Safe Haskell is the restrictions that using the safe language imposes. In the case of the restricted extensions, the FFI, `Data.Typeable` and Overlapping Instances, we believe the modified behaviour is an appropriate solution. However it is possible that rather than simply disabling Generalised Newtype Deriving, RULES and Template Haskell a more flexible solution could be found, restricting their behaviour instead of disabling the extension entirely. Generalised Newtype Deriving will be fixed in the future but perhaps there is a safe, restricted form that RULES and Template Haskell could operate in instead of the current all or nothing design.

The other limitation of Safe Haskell is that of resource exhaustion. Many practical uses of Safe Haskell for securing untrusted code will need to handle attacks designed to consume CPU and memory. At this time the only solution we are aware of is resorting to OS, process-level resource limits. This is a fairly heavyweight solution and we are interested in pursuing a language-level approach that would allow code to have resource limits applied within the same address space, either at the function or thread granularity.

8.1 Compilation safety

When a server compiles some untrusted code, the server must be sure that the act of compilation itself does not compromise the server in some way. This is effectively another security policy and moreover one that is a property of the tools rather than the language. As such we have deliberately separated it from the concept of the safe language.

Our implementation of Safe Haskell does not currently address the issue of safe compilation, although we recognise its importance. One simple solution available today is to compile untrusted code in a sandbox, such as a jail or virtual machine. Alternatively we could identify language and tool features that are a security threat and disable them. We believe that disabling the following features is sufficient for safe compilation with GHC:

- Template Haskell, Quasiquotation and the `ANN` pragma, all of which execute code at compile-time.
- The C preprocessor (CPP) and user-defined preprocessors, both of which may read unauthorised files during compilation.

However, we note that GHC is often not the only tool involved in compilation; packages of code may come with their own build systems which also need to be trusted. Cabal packages [10] for example can provide their own executable build systems in the form of a Haskell `Setup.hs` module and typically these build systems need to be able to do IO because they read and write files. Achieving safe compilation without an OS-level sandbox in these circumstances is likely to be difficult.

Finally, we also note that causing GHC to diverge or for the compilation to take an exponential amount of time is fairly easy to achieve. (Even Hindley-Milner type inference is known to have worst-case complexity exponential in the program size.) This is not a serious problem: while it may be used for a denial of service attack against a system, there is no way to abuse this to compromise the guarantees offered and a compilation time-out represent a simple and viable solution.

9. Related work

The basic idea of Safe Haskell, that of a safe subset of a language, is not a new one. Several other attempts in this space exist. There are some important differences and contributions we believe Safe Haskell makes compared to these works.

The major difference between Safe Haskell and other works is that Safe Haskell is deployed within the Haskell language itself, this is, as a feature of GHC, the dominant Haskell compiler. Safe Haskell works with the existing language and indeed builds on existing trends and coding styles in the community. A different set of tools or a slightly different language is not required. Unlike all other work, Safe Haskell also tracks module safety at every stage and for every compilation, whether or not the user is explicitly using Safe Haskell. This allows for the design of trustworthy modules and safe inference, both of which increase the ability for Safe Haskell to function within the existing Haskell language. Indeed this is perhaps our biggest contribution to the space: we allow more flexible use cases than all or nothing. For many users, our hope is that they will never even have to use Safe Haskell but that the in-

ference ability will still provide benefits for them and users of their code.

Another key difference is the flexibility and expressiveness of the language we are securing. We chose Haskell because we believe it is uniquely suited for implementing secure systems. The clear distinction between impure, potentially dangerous functions and pure ones through the `IO` monad allows for security policies to be expressed and reasoned about easily. When combined with the ability to implement alternatives to `IO`, like `RIO` or `LIO`, this gives users a powerful way build their own security mechanisms.

9.1 Modula3

Modula3 [1] is a research language developed in the 1980's that has a concept of a safe and unsafe module. Modula3 has some strong similarities to Safe Haskell and indeed we derived inspiration from its design. Modula3, unlike Haskell, separates a module's interface from its implementation, making interfaces first class. Both modules and interfaces can be marked as safe or unsafe, with safe being the default. Marking as unsafe enables access to certain unsafe operations in the language such as unchecked type casts. Safe modules cannot import or export unsafe interfaces, however, unsafe modules can export safe interfaces. Modula3 doesn't refer to this latter case as a trustworthy module but the concept is the same. An unsafe module that exports a safe interface is precisely the definition of a trustworthy module in Safe Haskell.

Safe Haskell differs from Modula3 however in its inclusion of the concept of trusted packages. The `-fpackage-trust` flag and the design of Safe Haskell to include the package system, not just the language, offers a level of control over trustworthy modules that Modula3 does not have. The inclusion of safe inference in Safe Haskell is another distinguishing feature compared to Modula3. Finally, Modula3 was designed from the start with the concept of a safe and unsafe boundary while Safe Haskell has undertaken to retrofit this to a mature language, building off existing idioms in the community.

9.2 Java

Since early on, Java has had a fairly comprehensive runtime security mechanism known as stack inspection [27], designed mainly for supporting Java applets. The system works by associating principals and privileges with classes based on how their code was loaded. For example, an applet code loaded over the web could be labelled with the `untrusted` principal and any local code running in the same JVM labelled with the `system` principal. A principal has access to a certain set of privileges that act as keys for performing certain sensitive operations. Accessing the file system is one such privilege; using the reflections API is another. Privileges can be enabled or disabled dynamically by Java code, allowing, for example, sandboxes to be built in which the only code enabling the filesystem privilege is the implementation of a minimal API that carefully checks all arguments. Java tracks privileges and principals dynamically and records the information in each stack frame. Before a sensitive operation takes place, the code implementing the operation must invoke the security manager, which walks the stack checking each frame until either a frame enabling the correct privilege is found, in which case the sensitive operation takes place, or an untrusted frame is found, in which case a security exception is thrown.

Stack inspection and Safe Haskell are each implemented at the core of a widely used language. Beyond that, the two have little in common. Stack inspection is a way of enforcing a particular class of security policies on the assumption that Java is type-safe. By contrast, Safe Haskell enforces type safety in Haskell, encouraging the use of types to express security policies but deliberately avoiding committing to any particular class of policies. While Safe Haskell uses static enforcement, stack inspection is dynamic, incurring an overheard of up to 9% [27]. Moreover, Stack inspection must be explicitly designed into the base libraries by invoking the security manager before every sensitive operation. This restricts enforceable policies to those with the same notion of sensitive operation, ruling out such policies as information-flow control, but also facilitates construction of policies that precisely match the sensitive operations.

9.3 JavaScript

The JavaScript language has also seen a fair amount of work with similar goals and techniques as Safe Haskell. Historically there has been the FBJS [6] and ADsafe [2] JavaScript frameworks developed by Facebook and Yahoo respectively. Both provide a mechanism for safely executing untrusted code, through language level sandboxing and a restricted API. These projects perform a source-to-source translation on JavaScript to ensure only a safe subset is used, much like the `Safe` pragma enforces. More recently the ECMA Standards Committee (TC39) developed a *strict mode* (ES5S) [4] and are developing a more restrictive sub-language of JavaScript called *Secure EcmaScript* (SES) [5, 23]. Both sub-languages make JavaScript a language more amenable to sandboxing with the SES environment being similar to our `Safe` pragma in the guarantees it provides. FBJS, ADsafe and SES all differ from Safe Haskell in that they focus exclusively on the security aspects while Safe Haskell also attempts to be broader than that. We explicitly designed Safe Haskell to fit into the existing language and be used in tandem with it as easily as possible. Also, while SES does offer a form of import control in the new *variable-restricted eval* function, that allows specifying an upper bound on what free variables some code can access, it has no notion of a module system and tracking of safety. This is simply due to the constrained nature of the browser that JavaScript is run in, which limits the scope of the problem.

9.4 Object capability languages

Finally, there is also a significant amount of work on securing programming languages with the object capability model [15, 16]. While this work is broader in focus, it overlaps with Safe Haskell in that many of the languages used were existing, widely used languages that needed to be modified in a manner similar to the Safe Haskell safe language, so the object capability model could be supported. Once the language is secure, implementing the capability model generally just involves careful API design, similar to the `RIO` technique we outlined. Examples of this work include the Emily [22] and E [20] programming languages, Google's Caja [17] (a variant of JavaScript) and Joe-E [14] (a variant of Java).

Joe-E is the most interesting example here as they designed a subset of Java that disables language features incompatible with the object capability model. Joe-E uses a static verifier to disable features such as native methods, reflection and global variables. While some of these could be dealt with using the Java security manager, they chose a stronger static guarantee. The changes to the language go much further than Safe Haskell due to the implementation of a security mechanism, not just type safety and module boundaries. Unlike Safe Haskell Joe-E assumes that all code is compiled in the safe variant and doesn't easily support a mixture of the safe and original variants of the language. No notion exists of module safety, safe imports or safe inference. This makes porting existing Java code to Joe-E a difficult task. The main advantage they gain from implementing Joe-E as a subset of Java is tool support and existing formal semantics. Safe Haskell has all of these but also the support (hopefully) of an existing community and a huge amount of code.

One interesting design choice of the Joe-E project is its work on the static verifier so that it can prove and enforce that methods are functionally pure [7]. They argue that this aids in the verification of security properties, a claim we agree with. This is largely achieved by designing the subset of Java such that purity can be determined directly from the type of a function. Obviously, this is a property Haskell has (somewhat) always had.

10. Conclusion

Safe Haskell offers a language extension that "hardens" the Haskell language by providing five properties: *type safety*, *referential transparency*, *strict module encapsulation*, *modular reasoning* and *semantic consistency*. It achieves this by providing both a safe subset of Haskell in which these properties hold and a trust tracking system for reasoning about the safety of modules and packages.

By closing the loopholes that exist in Haskell and making safety and trust first class, Safe Haskell makes it possible to confine and safely execute untrusted code. By leveraging Haskell's type system, we can build a variety of security mechanisms on top of the Safe Haskell foundation. We demonstrated this by building an online Haskell interpreter that executes possibly malicious code provided by unknown users.

We implemented Safe Haskell in GHC and it has been shipping since version 7.2. The implementation is capable of inferring the safety label of a module, effectively transitioning a large body of existing code to the safe subset. Upon evaluation we found that around 27% of existing code is already compatible with the safe subset and ready for use today by untrusted code.

Have we closed *all* the loopholes? We can offer no formal guarantee, because GHC is, necessarily, a complex beast in which security is in tension with functionality. Security mechanisms are, in the end, *always* ways to increase confidence and make attacks harder, rather than absolute guarantees. Seen in this light, Safe Haskell takes a useful step in the right direction.

Safe Haskell also informs GHC's development process: a bug that violates the Safe Haskell guarantees is a security bug and should be treated accordingly. Lastly, we believe that Safe Haskell is a valuable contribution to the Haskell language, because it will promote good practice amongst developers and clarify the boundaries between safe and unsafe code.

Acknowledgments

We thank Amit Levy and Deian Stefan for their help with writing the paper. We also thank the anonymous reviewers for their valuable feedback. This work was funded by the DARPA Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4088 and by a gift from Google.

References

[1] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). In *Technical Report*. Systems Research Center, Digitical Equipment Corporation, 1989.

[2] D. Crockford. Adsafe: Making JavaScript safe for advertising. `http://adsafe.org/`, 2008.

[3] C. Done. TryHaskell: An interactive tutorial in your browser. `http://tryhaskell.org/`, 2012.

[4] ECMA. ECMA-262: *ECMAScript Language Specification*. Fifth Edition, 2009.

[5] ECMA. Ses: *Secure EcmaScript Language Specification*. `http://wiki.ecmascript.org/doku.php?id=ses:ses`, 2009.

[6] Facebook. Fbjs (Facebook JavaScript). `http://developers.facebook.com/docs/fbjs/`, 2012.

[7] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *Computer and Communications Security*. ACM, 2008.

[8] GHC Bug Tracker. Newtypes and type families combine to produce inconsistent fc(x) axiom sets. `http://hackage.haskell.org/trac/ghc/ticket/1496`, 2007.

[9] W. Harrison. Achieving information flow security through precise control of effects. In *Computer Security Foundations Workshop*. IEEE Computer Society, 2005.

[10] I. Jones. The Haskell cabal, a common architecture for building applications and libraries. In *Trends in Functional Programming Symposium*, 2005.

[11] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Languages Design and Implementation Workshop*. ACM SIGPLAN, 2003.

[12] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Computer Security Foundations Workshop*. IEEE Computer Society, 2006.

[13] S. Marlow *(editor)*. Haskell 2010 language report. 2010.

[14] A. Mettler, D. Wagner, and T. Close. Joe-e: A security-oriented subset of Java. In *Network and Distributed System Security Symposium*. Internet Society, 2010.

[15] M. Miller. Robust composition: Towards a unified approach to access control and concurrency control. In *Ph.D. Dissertation*. Johns Hopkins University, 2006.

[16] M. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. In *Technical Report*. Johns Hopkins University, 2003.

[17] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `http://code.google.com/p/google-caja/`, 2008.

[18] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information flow security in Haskell. In *Haskell Symposium*. ACM SIGPLAN, 2008.

[19] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, 2011.

[20] M. Steigler and M. Miller. How Emily tamed the Caml. In *Technical Report HPL-2006-116*. HP Laboratories, 2006.

[21] D. Stewart. Lambdabot. `http://hackage.haskell.org/package/lambdabot`, 2012.

[22] M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *Creating, Connecting and Collaborating through Computing Conference*. IEEE Computer Society, 2007.

[23] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript APIs. In *Security and Privacy Symposium*. IEEE Computer Society, 2011.

[24] D. Terei and GHC Team. Hackager: A Hackage testing tool. `http://hackage.haskell.org/trac/ghc/wiki/HackageTesting`, 2012.

[25] T.-c. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Computer Security Foundations Symposium*. IEEE Computer Society, 2007.

[26] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *International Conference on Functional Programming*. ACM SIGPLAN, 2004.

[27] D. Wallach. A new approach to mobile code security. In *Ph.D. Dissertation*. Princeton University, 1999.

[28] S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Principles of Programming Languages Symposium*. ACM SIGNPLAN, 2011.