Inferring Required Permissions for Statically Composed Programs

Tero Hasu, Anya Helene Bagge, and Magne Haveraaen

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~{tero,anya,magne}

Abstract. Permission-based security models are common in smartphone operating systems. Such models implement access control for sensitive APIs, introducing an additional concern for application developers. It is important for the correct set of permissions to be declared for an application, as too small a set is likely to result in runtime errors, whereas too large a set may needlessly worry users. Unfortunately, not all platform vendors provide tools support to assist in determining the set of permissions that an application requires.

We present a language-based solution for permission management. It entails the specification of permission information within a collection of source code, and allows for the inference of permission requirements for a chosen program composition. Our implementation is based on Magnolia, a programming language demonstrating characteristics that are favorable for this use case. A language with a suitable component system supports permission management also in a cross-platform codebase, allowing abstraction over different platform-specific implementations and concrete permission requirements. When the language also requires any "wiring" of components to be known at compile time, and otherwise makes design tradeoffs that favor ease of static analysis, then accurate inference of permission requirements becomes possible.

Keywords: language-based security, platform security architectures, security management, software engineering

1 Introduction

Permission-based security models have become commonplace in real-world, consumer-faced operating systems. Such models have been adopted mostly for mobile OS platform security architectures, partly because smartphones are high-utility personal devices with privacy and usage cost concerns (regulations and business models have also driven adoption [21]). Smartphones are also natively third-party programmable (by our definition), and the wide consumer awareness of "app stores" has made it almost an expectation that applications (or "apps") are available for installation in large numbers. While some smartphone platforms

(such as iOS[1] and Maemo) rely on app store maintainers to serve as gatekeepers against malicious (or maliciously exploitable) apps, many others (such as Android, BlackBerry 10, and Windows Phone) have permission-based security to restrict the damage that such apps might cause. Sole reliance on gatekeepers has the drawback that "side-loading" of apps from another source is then more likely to be prevented by the platform vendor (as is the case with iOS[2]).

A number of different terms are being used for essentially the same concept of a permission. By our definition a *permission* is something that is uniquely named, and something that a program (or rather its threads of execution) may possess. Possession is required for a program to be allowed to take certain actions (typically to call certain system APIs), or perhaps even to be the target of certain actions (e.g., an Android app may not receive certain system messages without the appropriate permissions [15]). A common reaction to an attempt to invoke an unallowed operation is to trigger a runtime error, although the concrete mechanisms for reporting such errors vary between platforms.

By the term *permission-based security model* we simply mean a security model in which access control is heavily based on permissions. We assume at least API access control such that different permissions may be required for different operations; i.e., there is finer than "all or nothing" granularity in granting access to protected APIs. With judiciously chosen restrictions for sensitive APIs a permission-based security model can serve as a central platform integrity protection measure. Such a model can also help permission-savvy end users (even if they are in the minority [16]) avoid leakage of private data and malicious exploitation of functionality.

Users, operators, and regulators all get some genuine benefit from platform security measures. Software developers, however, tend to only be inconvenienced by them, unless their software specifically requires functionality that platform security happens to provide. There are restrictions in what features can be had in an app, and how apps can be deployed (during a test/debug cycle, or in the field). This can even motivate the maintenance of multiple variants of an app [18] depending on what permissions are grantable for which distribution channel.

For most platforms the permissions required by an application must be declared. Writing the declaration may not in itself be difficult, but permission requirements are sometimes poorly documented [15], and keeping permission information up to date is an extra maintenance burden. The burden can be significant particularly for applications [18] that both exercise many sensitive APIs, and also have variants with different feature sets.

We present an approach for inferring permission requirements for programs constructed out of a selection of components in a permission-annotated codebase. While it takes effort to annote all sensitive primitives with permission information, the up-front cost is amortized through reuse in new program com-

[1] In iOS 6, there is a small set of privacy-related permissions with application-specific settings. Developers need not declare the required permissions.

[2] As of early 2013, end-user installation of iOS applications is only allowed from the official vendor-provided App Store.

positions. We have implemented the approach as one use case for the research language Magnolia, designed to be statically analyzable to the extreme. Magnolia avoids dynamic features, but has extensive support for static "wiring" (or linking) of components. We argue that these characteristics combine to facilitate permission inference without undue restrictions on expressivity. Magnolia also supports cross-platform code reuse, as its interface and implementation specifications allow for declaration of permission information in such a way that different platform-specific concrete permissions can be handled in an abstract way.

Magnolia is source-to-source translated into C++, and hence can be used to target platforms that are programmable in C++, including most smartphone platforms.[3] Translation to a widely deployable language is an important part of the overall portability picture, and also a possibility to abstract over differences in implementations of said language. Cross-platform libraries and Magnolia's support for interface-based abstraction help with the API aspect of portability. A third aspect is support for integration with platform vendor provided tools, which remains as future work in the case of Magnolia.

Maintaining permission information together with source code should result in better awareness of possible runtime permission failures when programming, and also allow for various automated analyses of the permission requirements of programs and program fragments. Such analyses, particularly when used in ways that affect the construction of software (e.g., due to analysis-based generation of permission declarations, or even code modifications), could also aid in the discovery of errors in app permission declarations or platform documentation.

While our focus is on permissions, some of the techniques presented apply not only to *right* of access, but more generally *ability* of access. E.g., from the point of view of error handling it matters little if a runtime failure is caused by lack of camera hardware, or lack of permission to access it. There are platform differences in whether requesting a permission will guarantee its runtime possession, and also in whether it is possible to similarly declare a (software or hardware) feature requirement so that availability of the feature will be guaranteed after successful installation. For instance, specifying `ID_REQ_REARCAMERA` in the manifest of a Windows Phone 8 app will prevent installation on devices without a back-facing camera [22]. Given the similarities between permissions and feature requirements we sometimes use the term *access capability* to imply access ability in a broader sense than that determined by permissions.

1.1 Contributions

The contributions of this paper are:

- We give a brief overview of permission-based security models of a number of current smartphone OSes, and survey the associated tooling (if any) for inferring required permission information for applications.

[3] Magnolia is not a *symbiotic language* (i.e., a language designed to coexist with another one), however, and there is nothing in Magnolia that would prevent its compilation into other languages. Still, the current implementation only targets C++.

- We present a language-based solution for declaring permissions for APIs and inferring permission requirements for programs. The solution allows for cross-platform programming by exploiting the host language's support for interface-specification-level abstraction over different implementations.
- We discuss static analysis friendly language design choices that favorably affect permission inference accuracy, and argue that some of the expressiveness cost of the resulting lack of "dynamism" can be overcome by flexible static composition.

To evaluate the presented solution we have implemented it based on the Magnolia language, and made use of it in a small cross-platform porting friendly application that requires access to some sensitive APIs. We have organized the app codebase to facilitate growing it to target multiple different platforms and feature sets, probably with different permission sets for different configurations.

2 Permission-Based Security Models in Smartphone Operating Systems

Below we list distinctive aspects of the permission-based security models of a number of current smartphone OSes (more wholesome surveys of the permission and security models of some of the same platforms exist [2, 21]). We also discuss any permission inference or checking tools in the associated vendor-provided developer offerings. We provide a side-by-side summary of permission-related details of the platforms in Table 1. Due to the newness of Tizen (no devices have been released as of early 2013) and the similarity of its and bada's native programming offerings, we opt to exclude Tizen (but not bada) from the table.

Android allows for the definition of custom "permissions". Permissions have an associated "protection level", with permissions of the "dangerous" level possibly requiring explicit user confirmation; hence a developer defining such a permission should also provide a description for it, localized to different languages [1]. A "signature" level permission does not have that requirement as it is automatically granted to apps signed with the same certificate as the app that declared the permission. No tools for inferring permissions for an app are included in the Android "SDK Tools" [1] as of revision 21.1. There are two third-party permission checkers capable of statically analyzing the permission requirements of Android apps. The tools are named Stowaway [15] and Permission Check Tool [30], and they both report on over/underprivilege wrt manifest-declared permissions. Their accuracy is discussed in Section 7.

bada 2.0 The Eclipse-based IDE of the bada SDK 2.0.0 [27] incorporates an "API and Privilege Checker" [26] tool that checks the project for privilege violations (an API requiring a privilege group is used, but the privilege group is not declared in the manifest) and unused privileges automatically during packaging, and optionally during builds. The tool is for *checking* privileges, and does not generate privilege group declarations for the manifest.

BlackBerry 10 (BB10) is notable in that (upon first running an app) a user may grant only a subset of the "permissions" requested in the corresponding "application descriptor file" [13], and it is then up to the app to react sensibly to any runtime failures caused by unpermitted operations. BB10 also has limited support for running (repackaged) Android applications, with a number of Android features and permissions being unsupported [12].

MeeGo 1.2 Harmattan access control makes use of traditional "credentials" including predefined Linux "capabilities", Unix UID and GID and supplementary groups, and file system permissions. Harmattan adds to these by introducing fine-grained permissions known as resource "tokens", as supported by the Mobile Simplified Security Framework (MSSF) [23]. Granting of credentials is policy-based, and consequently (as of early 2013 and Harmattan version PR1.3) app credential information is not shown to the user, either in the app Store or under installed Applications. The `aegis-manifest` tool performs static analysis of binaries and QML source. It generates a manifest file listing required credentials for a program, but may fail in exceptional cases. Dynamically determined loading (e.g., via `dlopen`) or invocation (e.g., via D-Bus) of code are possible causes for the static scanner failing to detect the full set of required credentials.

Symbian v9+ Symbian OS has had a "capability-based security model" since version 9 [19]. It is unusual in that both executables and DLLs have "capabilities". A process takes on the capabilities of its executable. Installation requires code signing with a certificate authorizing all the capabilities listed in any installed binaries; a self-signed certificate is sufficient for a restricted set of capabilities. Any loaded DLLs must have *at least* the capabilities of the process. There is a "Capability Scanner" plug-in for the Eclipse-based Carbide.c++ IDE that ships with some native Symbian SDKs; the plug-in is available starting with the Carbide.c++ release 1.3 [24]. The scanning tool presents warnings about function calls in a project's codebase for which capabilities are not listed in the project definition file. The tool is only able to *estimate* the required capabilities.

Tizen 2.0 The Tizen 2.0 SDK [29] release introduced a C++ based native application framework, which appears to have bada-derived APIs. The permissions in Tizen are called "privileges"; the set of permissions (and their naming) in Tizen differs from those of bada. Privileges are specified in a manifest file in an installation package, and there is no tool support for automatically inferring and generating the privilege requests. However, as with bada, the Tizen SDK includes an "API and Privilege Checker" [28] tool for checking for potential inconsistencies between specified privileges and APIs being used in an application. The tool may be enabled for automatic checks during builds or code editing, and it may detect either under or overprivilege.

Windows Phone 8 (WP8) has a security model in which the kernel is in the "Trusted Computing Base" "chamber", and where OS components, drivers, and apps are all in the "Least Privilege Chamber" (LPC) [20]. Software in the latter chamber may only directly invoke relatively low-privilege operations, and only when in possession of the appropriate "capabilities". All capabilities

	Android	*bada*	*BlackBerry 10*	*MeeGo 1.2 Harmattan*	*Symbian v9+*	*Windows Phone 8*
permissions:	open-ended set of "permissions"	predefined set of "privilege groups"	open-ended set of "permissions"	open-ended set of resource "tokens"	predefined set of "capabilities"	predefined set of "capabilities"
permission categories:	"normal", "dangerous", "signature", and "signatureOrSystem"	"Normal", "System"	N/A	N/A	"User", "System", "Restricted", "Device Manufacturer"	"Least Privilege Chamber"
auth:	depending on permission: automatic, user approved (all or nothing), signed by authority, or preinstalled	by vendor at time of publishing, based on *developer* "privilege level"	user approval; user may only grant a subset of requested permissions	by installer, depending on software source and policies declared in software packages	user approved (all or nothing), developer signed, identity-verified developer signed, or vendor approved	user approval (all or nothing)
assignment request:	recorded in a manifest in installation package	recorded in a manifest in installation package	recorded in a manifest in installation package	recorded in a manifest in installation package	specified in project definition; recorded in binary	recorded in a manifest in installation package
inference by tools:	Stowaway (3rd party), Permission Check Tool (3rd party)	"API and Privilege Checker"	none	aegis-manifest	"Capability Scanner"	none (for WP8 – "Store Test Kit" for 7.1)

Table 1. Smartphone platform permissions and tools support.

are user grantable, and the requested capability set of each app is disclosed in Windows Phone Store; some capability requirements are displayed more prominently than others. "Hardware requirements" may also be specified, and an app is not offered for phone models not meeting the requirements. The Windows Phone SDK 8.0 does not contain capability detection tools for apps targeting WP8[4], nor (as of early 2013) are such apps programmatically capability analyzed during Store submission [22].

3 The Magnolia Programming Language

Magnolia [7] is a research language that aims to innovate in the area of reusability of software components. Safe composition of reusable components requires

[4] Windows Phone SDK 8.0 has a Visual Studio IDE integrated "Store Test Kit" that may be used to inspect a Windows Phone OS 7.1 targeting app and list the capabilities required by it. Windows Phone OS 7.1 is not natively programmable by third parties, and hence not a smartphone OS per our definition.

strict specification of component interfaces—sometimes referred to as APIs (application programmer interfaces) if semantic content is implied. A description of an API in Magnolia is given using the `concept` construct; a `concept` declaration can be thought of as an incomplete requirements specification. It specifies one or more abstract `type`s, some operations on those types, and the behavior of those operations (in the form of axioms). Each `concept` may have multiple `implementation`s that provide data structures and algorithms that satisfy its behavior. Each `implementation`, in turn, may satisfy multiple `concept`s.

One kind of operation that may be defined in Magnolia is a `procedure`. A `procedure` has no return values, but may modify its arguments according to specified *parameter modes* [6]. Legal parameter modes include `obs` (observe; the argument is read-only), `upd` (update; the argument may be changed) and `out` (output; the argument is write-only) [8]. A simple `procedure` that only outputs to a single parameter may equivalently be defined in a more "sugary" form as a `function`, and regardless of choice of declaration style, invocations to such operations may appear in expressions.[5] The keyword `call` is used to invoke an operation as a statement. A `predicate` is a special kind of function yielding truth values, and taking zero or more appropriately typed expressions as arguments. A `predicate` application as well as `TRUE` and `FALSE` are *predicate expressions*, and more complex predicate expressions are built using logical connectives.

The notion of *partiality* of an operation, meaning that the operation is not valid for all values that its parameter types could take, is central to Magnolia. Such a restriction can be specified for an operation. In the API it takes the form of a `guard` [4] with a predicate expression, which may include invocations to `function`s and `predicate`s. The more fine-grained notion of *alerts* [5] is the corresponding partiality notion in implementations. Alerts are an abstraction over `pre`/`post`conditions and error reporting, and each partial function is tagged with a list of `alert` names and the corresponding conditions that trigger the alerts. The set of defined alert names is user extensible and partially ordered, possible to organize as a directed acyclic graph.

```
alert CameraAccessAlert;
alert NoCamera <: CameraAccessAlert;
alert NoAccessToCamera <: CameraAccessAlert;

predicate deviceHasCamera() = Permission;
procedure takePicture(upd w : World, out p : Picture)
  alert NoCamera unless pre deviceHasCamera()
  alert NoAccessToCamera if throwing PermissionDenied
  alert NoAccessToCamera if throwing CameraInUse;
```

Here the alert names `NoCamera` and `NoAccessToCamera` are specialisations of the alert name `CameraAccessAlert`. The procedure `takePicture` has three possible error behaviors. The precondition test calling the predicate `deviceHasCamera` checks whether the device has a camera; if not, it would not be meaningful to use the

[5] In Magnolia, an expression always yields a single value; i.e., there are no multi-valued expressions such as `(values 1 2)` in Racket.

procedure. The two other conditions have the same alert name, and are triggered by the procedure implementation throwing one of two exceptions.[6]

A `program` is a special implementation in that its operations are made available as "entry points" to a piece of software that is composed in Magnolia. The Magnolia compiler translates Magnolia code into C++ source code, and produces a command-line interface wrapper for the `program` through which the exported operations may be invoked.

Due to Magnolia's explicit static linking of components (as declared in source code), all data structures and algorithms corresponding to a `program`'s types and operations (respectively) are known at compile time. Programs are statically typed, and there is no subtyping or dynamic dispatch (as e.g. in the case of C++ `virtual` functions). There are also no first-class functions (or even function pointers) to pass by value for parameterizing operations at runtime; any such parameterization must be done statically by specifying concrete operations used to implement a concept.

The static nature of Magnolia means that the actual target of an operation invocation appearing in program code is always statically known. Due to this it is possible to tell whether calls to a given operation appear in a given program composition, and any definitions for operations that have no invocations may be dropped for purposes of optimization or full-program analysis. Still, even in Magnolia's case it is generally not possible to tell if an operation appearing in a program actually gets invoked, as relevant facts about program runtime state or how far execution gets to proceed are generally not known at compile time.

4 Language Support for Permissions

Here we design a way to model permissions (and more generally, access capabilities) in Magnolia. As we prefer to keep Magnolia's core language simple, again for ease of analysis, we want to avoid feature-specific language extensions where possible. In this case we can do so by mapping permissions onto the Magnolia alerts system. The syntax may not always be as convenient as it could be, but that could be fixed through superficial syntactic transformations; we do not consider alternative syntaxes here.

The execution of a program consists of operations on the program state, and we want to be able to determine the permission requirements of all operations appearing in Magnolia code. To allow for this the permissions must either be declared, or it must be possible to infer them based on the implementation of the operation (i.e., its body). Magnolia currently allows an operation to be implemented either in Magnolia or in C++; for the former we can infer permissions by examining the language, but not for the latter. Any permission requirements for C++ operations will therefore have to be declared.

[6] In real-world code we might want different alert names to distinguish between errors of a transient (`CameraInUse`) and permanent (`PermissionDenied`) nature. On most platforms application permissions are fixed at install time.

Permission-protected operations are associated with requirements, i.e., preconditions, as dictated by the platform APIs. We can state the preconditions as `alert`s with predicate expressions, noting that a permission restriction gives us two separate concerns: (1) we want to know of the permission requirement so that we can request the permission, and hence try to prevent runtime errors; and (2) we want to be able to handle any related errors. For case (1) we want platform-specific permission names, while for case (2) we would like abstract, platform-agnostic *error* names, probably relating to the operation. The example in Section 3 had the latter kind of names, namely `NoCamera` and `NoAccessToCamera`.

For storing platform-specific permissions we essentially just want to have the predicate expressions as named properties of operations. Had we support for convenient scripting of compiler-assisted queries we would not necessarily require fixed, predefined naming, but might rather choose any descriptive name to use as a search key to find the relevant expressions. The built-in support for permission inference in Magnolia currently uses the name `RequiresPermission` for this purpose (as suggested in Section 6, it might sometimes be desirable to use other names). We use `RequiresPermission` to "tag" permission preconditions, and each permission appearing in a precondition is defined as a "dummy" `predicate`.

As such predicates merely represent static properties, they are not intended to actually trigger an `alert` at runtime. This can be ensured by treating `RequiresPermission` as special and not inserting a precondition check for it. A more general alternative is to define the predicates as `TRUE`, leaving any generated check as dead code. On most platforms we can assume that the program is only started if the declared permissions have been granted, but there may be reasons for not requesting all inferred-as-required permissions. Permission-related precondition violations are thus possible, and we want them trapped as declared for their platform-agnostic `alert`s. It may be more efficient to capture any platform-specific runtime "permission denied" error than to actually implement a sensible predicate that checks for possession of the associated permission.

The Magnolia compiler supports scavenging a `program` for its operations (which, as mentioned in Section 3, are known in Magnolia) and respective permission requirements, provided the operations' permissions are specified as suggested above. (This approach also generalizes to other access capabilities, e.g. Windows Phone hardware requirements.) The result is conservative, but can only err on the side of too many permissions, assuming correct annotations. One source of inaccuracy is the currently indiscriminate inspection of all operations. Any dead code elimination done by the compiler happens later in the pipeline; such optimization would be beneficial, particularly if data-flow sensitive.

The second source of inaccuracy comes from the way we build the result. Perhaps the most accurate way to represent the result would have been as a single predicate expression such as `BLUETOOTH() && CAMERA() && (ACCESS_COARSE_LOCATION() || ACCESS_FINE_LOCATION())`, built as a collation of the relevant predicate expressions. Currently, however, we just build a set of permissions such as {`BLUETOOTH`, `CAMERA`, `ACCESS_COARSE_LOCATION`}. This may produce suboptimal re-

sults, as concrete choices must be made between logical alternatives. Our current implementation produces a set, and picks the left choice from OR-ed permissions.

Platform-provided sensitive operations typically require a fixed set of permissions, but there are many exceptions that motivate allowing the use of logical expressions to at least *specify* permission requirements, even if we do not always make optimal use of the specification. Let us consider the `LocationManager` class of Android OS. Its `getLastKnownLocation(String)` method requires either `ACCESS_FINE_LOCATION`, or at least `ACCESS_COARSE_LOCATION`, depending on the "location provider" specified as the sole argument. The `NETWORK_PROVIDER` supports both coarse and fine grained positioning, and no `SecurityException` should get thrown as long as either permission has been requested (and granted). If we implement a network positioning specialized version of the operation—perhaps named `getLastKnownNetworkLocation`—then we may declare:

```
procedure getLastKnownNetworkLocation(upd w : World, out l : Loc)
  alert RequiresPermission unless pre ACCESS_COARSE_LOCATION() ||
      ACCESS_FINE_LOCATION()
  alert LocationAccessNotPermitted if throwing SecurityException
  alert IllegalArgument if throwing IllegalArgumentException
  alert NotFound if post value == null;
```

We are using a platform-agnostic `LocationAccessNotPermitted` alert to allow permission failures to be handled portably. The Android-specific permissions we are stating as a predicate expression tagged with `RequiresPermission`. Other possible errors for the operation are also mapped to alerts to allow handling.

For other platforms we would probably require a different (native) implementation of the operation, also with different error-to-alert mappings declared similarly to the above. E.g., on Windows Phone a `UnauthorizedAccessException` typically gets thrown on permission errors, whereas on Symbian one can generally expect a Symbian-native *leave* (a form of non-local return) with the error code `KErrPermissionDenied`. Interestingly, there are APIs (such as those of the Qt cross-platform application framework) that have been ported to different platforms, but which still necessarily have platform-specific permission requirements. With such APIs one could have a single (native) implementation but multiple Magnolia declarations (with different `alert` clauses).

5 Experience with Application Integration

For trying out the solution we created a small software project named *Anyxporter* (Any Exporter) [11], with the goal of building a codebase that would serve as a basis for creating various programs for exporting PIM (personal information manager) data in different (probably textual) formats. We chose the PIM exporting theme for exercising permissions as: (1) there are a number of different data sources, possibly requiring different permissions; (2) different storage/transmission options for exported data would likely require further permissions; and (3) the idea of building a "suite" of programs should allow us to keep the permission requirements of each individual program reasonably small, which may make a

user feel safer in installing a given variant (since the program does not ask for permissions to do anything other than what the user wants done).

Anyxporter currently includes only one proper PIM *data source*, for reading contact data. Its implementation requires the Qt Mobility Contacts API [25]. Said API is implemented [25] at least for Symbian (S60 3rd Edition FP1 and later), Maemo 5, and Harmattan, and also for Qt Simulator for testing purposes (without real contact data). For targets for which the API is not available, we have also implemented a "mock" data source that yields fixed contact data, and this data source has proved useful in testing other components of the software.

Of the targets supported by Qt Mobility Contacts, Symbian and Harmattan have permission-based security models, and our discussion here focuses on them. On Harmattan using the Qt API to *read* contact data requires the `TrackerReadAccess`, `TrackerWriteAccess`, and `GRP::metadata-users` credentials, whereas on Symbian only `ReadUserData` is required; clearly, the Symbian implementation of the API is better in respecting the principle of least privilege.

The default output option is to save to a file, which for a suitably chosen filesystem location requires no manifest-declared permissions either on Symbian or Harmattan. Anyxporter also has initial support for HTTP POST uploads of output files, implemented in terms of Qt 4.8 networking. Qt 4.8 is mostly unavailable on our example platforms, but Internet access generally requires no credentials on Harmattan, and the `NetworkServices` capability on Symbian.

Formatting of data for output is done using Lua scripts, and we currently include an XML formatting option for contact data. A Lua virtual machine (VM) instance is used as the intermediate representation (IR) between the different input and output options; in principle, data of the same kind (e.g. contact data) could have the exact same Lua object representation, regardless of concrete data sources and output formatters. Through careful choice of enabled Lua libraries we are preventing Lua code from doing anything other than "pure processing"; it cannot access platform APIs or the file system, and hence should require no permissions (or analysis for inferring permissions) on any platform.

The various library components of the app, such as file system interface, contact data source and Lua script interface, are specified by `concepts`. The main app code is programmed against these concepts, so that it is independent of the target platform. The app code is unaware of the exact nature of the permissions, though it may make use of and handle generic *permission denied* alerts.

Each library component has multiple implementations, one for each supported platform, with each implementation specifying platform-specific permissions. For example, the plain streams-based file system interface uses the following permission predicates:

```
predicate CXX_FILE_CREATE() = Permission;
predicate CXX_FILE_WRITE() = Permission;
predicate CXX_FILE_READ() = Permission;
predicate CXX_FILE_DELETE() = Permission;
```

A particular version of the app is built by composing the main app code with the platform-specific library implementations:

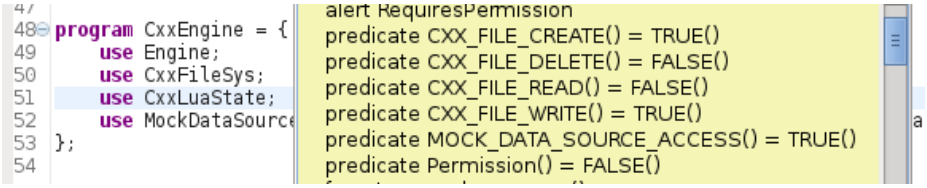

Fig. 1. Hover information for a program in the IDE shows which permissions are enabled and disabled.

```
program CxxEngine = {
  use Engine; // application logic
  use CxxFileSys; // generic C++ versions of the library components
  use CxxLuaState;
  // use the 'mock' data source
  // the data source mapper will apply 'exportEntry' to each data entry
  use MockDataSourceMapper[map => mapDataSource, Data1 => File, Data2 =>
      LuaState, f => exportEntry];
};
```

Our system collects all the permissions used by `CxxEngine`, defines the value of the relevant predicates to be `TRUE` (and the predicates for the unused permissions to be `FALSE`), and then outputs the permission list in a text file, together with the C++ code for the program. Figure 1 shows an IDE display with the inferred permission requirements.

6 Problematic Permission Requirements

It is a Magnolia philosophy that incomplete specifications are okay, and that specifying as much as is convenient is likely to give a good return for effort. Documented platform permission requirements are generally straightforward for individual operations, and it is unfortunate if they do not directly translate into code, as one must then expend effort to considering how to best specify them without harmful inaccuracies. There are real-world permission requirements whose accurate and convenient specification challenges our design.

It is not uncommon for the permission requirements of a platform operation to depend on its arguments. Such requirements *can* be specified as a predicate expression for an `alert`, as shown by the example below. However, as argument values are generally not statically known, the operation is no longer guarded by a static predicate expression. Any permission analysis trying to determine the permission requirements of a program will then require a policy regarding how to translate such expressions to static ones without underprivilege or too much overprivilege. Perhaps a better alternative is to (where possible) divide the operation into multiple ones with static predicate expressions. We did so in a similar example in Section 4 by defining a location provider specific operation for a provider known to support coarse-grained positioning.

```
procedure getLastKnownLocation(upd w : World, out l : Loc,
                               obs p : Provider)
  alert RequiresPermission unless pre ACCESS_FINE_LOCATION() ||
   (supportsCoarse(p) && ACCESS_COARSE_LOCATION());
```

We have discussed declaring different permissions for different platforms, but there are also permission differences between different releases of the same platform. On Android, the permissions for some operations have changed over time due to subtle and innocuous code changes [3] in their implementation. As such changes tend to only affect relatively few APIs and operations, it may be inconvenient to have to give separate `implementation` declarations in these cases. One possible, pragmatic solution may be to give different `alert` clauses for different platform releases. For example, we might generally specify `AndroidPerm` alerts for Android, but in some [3] cases use release specific alerts:

```
alert AndroidPerm8 <: AndroidPerm; // Android 2.2 (API level 8)
alert AndroidPerm9 <: AndroidPerm; // Android 2.3 (API level 9)
procedure startBluetoothDiscovery(upd w : World)
  alert AndroidPerm8 unless pre BLUETOOTH()
  alert AndroidPerm9 unless pre BLUETOOTH() && BLUETOOTH_ADMIN();
```

7 Related Work

Most of the literature on permissions is focused on Android, while our approach is to exploit the abstraction facilities of Magnolia in order to create platform-agnostic solutions. In Section 2 we already mentioned Stowaway [15] and Permission Check Tool [30], tools for analyzing the permission requirements of Android apps statically. As both tools are geared towards checking already declared permissions against code, the issue of deriving a concrete set of permissions to declare is perhaps less prominent; as explained in Section 4, the Magnolia compiler requires a policy for resolving logical permission expressions into sets.

Both Stowaway and Permission Check Tool resort to heuristics due to complexities of language and execution environment; heuristics-demanding complexities relating to language should not arise in the context of Magnolia. Stowaway's analysis appears more comprehensive than that of Permission Check Tool in that it attempts to handle reflective calls and Android "Content Providers" and "Intents". Magnolia has no reflective calls, and we propose that permissions be declared for *all* external-facing interfaces. Permission Check Tool works by analyzing source code using Eclipse APIs, whereas Stowaway takes Dalvik executable (DEX) files as input; the Magnolia ideal is to have programmable language infrastructure for custom analyses of semantically rich source code.

The Stowaway authors tackled poor platform documentation by determining Android 2.2 API permission requirements through API fuzzing. The PScout [3] tool has been found to discover more complete Android OS permission information. It performs a static reachability analysis between Android API operations and permission checks to produce a set of required permissions for each operation. Like our permission inferrer, PScout does path-insensitive analysis on

source code. PScout's policy for "expression-to-set translation" is to take the union of all appearing permissions, which is more conservative than ours.

PScout has been used to extract permission specifications for multiple versions of Android. We are not aware of such analyses for other OSes, and problems of poor API documentation are compounded for cross-platform programming. With a suitably accurate and complete permission map available for a platform, one might imagine annotating a primitive with its set of sensitive operations rather than its permission requirements, allowing for the latter to be inferred.

The kind of variability imposed by access capabilities is commonly handled using feature models [9, 10]. As shown in Section 4, access capabilities are associated with specific operations of an API, thus letting us use the alerts system of Magnolia for modeling their variability.

nesC [17] is a prominent example of a programming language with a programming model that is similarly restricted as that of Magnolia. Like Magnolia, nesC does static wiring of components so that types and operations become known at compile time; nesC even performs static component *instantiation* to avoid the overhead of dynamic memory management. The static nature of the language gives rise to a number of possibilities for accurate program analysis. E.g., the nesC compiler itself performs static whole-program analysis to detect data races. As nesC code is amenable to such analyses and the language also features interface-based abstraction support, we believe it would be a suitable substrate for a cross-platform permission inference solution. However, permissions are not applicable to TinyOS programming, which presently is nesC's primary domain.

As demonstrated by tools such as VCC [14], even unsafe languages (such as C) can be made static analysis (or verification) friendly with a suitably structured programming style and the addition of semantic information in the form of annotations. Additional annotations could also be used for permissions. Annotating an existing language is a valid implementation strategy for an analyzable language, with the advantage of avoiding another, full language layer. Magnolia's ground-up design for analyzability is likely cleaner, and the language can also be used merely as a tool for assembling programs out of C++ components.

8 Conclusion

Permissions are among the nuisances that software developers have to deal with. Language-based technology cannot lift access control restrictions, but it can help manage them, and reduce the chance of uncleanly handled permission errors occurring. Appropriate tools support enables automated analyses for determining a set of permissions that (if granted) will mean that no permission-caused runtime failures will occur. Suitable language can also help handle runtime failures in a portable manner, using abstract, concept or operation specific (not platform specific) permission failure reports and handlers.

We have presented such language and tools support. Our design relies on the base language taking care of: enforcing a programming style that does not prevent accurate static reachability analysis; and encouraging interface-based

abstraction. Mere ability to declare permission information in a language is not special, as many languages (e.g., Java and Python) even support annotations as a way to attach custom attributes to declarations.

In Magnolia, the base language of our implementation, we can use core language such as `predicate`s and `alert`s to express permission conditionality and errors. Cross-platform interfaces may be exposed as `concepts`, and different `implementation`s and/or `alert` declarations may be used to express platform differences. Coupled with tooling, code analyses (and also transformations) can be performed based on such declared information and what it implies.

In Magnolia, "dynamism" can only be allowed in a controlled way for correct permission analysis, and even then only outside the language. Analyzable, "static" language can be sugar-coated with convenient syntax, but certain familiar constructs are not directly transferable to Magnolia; e.g. a "traditional" higher-order `map` operation cannot be defined as functions cannot be passed as (runtime) arguments. Magnolia therefore carries some cost to expressiveness and developer familiarity, but offsets that by offering rich compile-time semantic information. Different language design tradeoffs could probably be made, while still allowing for accurate cross-platform permission inference. We see value in exploring awareness creating and preventative measures against potential software failures, whether caused by access control restrictions or other reasons.

Acknowledgements. We thank the anonymous referees for insightful comments on a draft of this paper. This research has in part been supported by the Research Council of Norway through the project DMPL – Design of a Mouldable Programming Language.

References

1. Android Open Source Project: Android Developers, `https://developer.android.com/`, retrieved May 2013
2. Au, K.W.Y., Zhou, Y.F., Huang, Z., Gill, P., Lie, D.: Short paper: A look at smartphone permission models. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. pp. 63–68. SPSM '11 (2011)
3. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: analyzing the Android permission specification. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 217–228. CCS '12 (2012)
4. Bagge, A.H.: Separating exceptional concerns. In: Proceedings of the 5th International Workshop on Exception Handling (WEH'12). pp. 49–51. IEEE (June 2012)
5. Bagge, A.H., David, V., Haveraaen, M., Kalleberg, K.T.: Stayin' alert: Moulding failure and exceptions to your needs. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06). ACM Press, Portland, Oregon (October 2006)
6. Bagge, A.H., Haveraaen, M.: Interfacing concepts: Why declaration style shouldn't matter. In: Ekman, T., Vinju, J.J. (eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09). Electronic Notes in Theoretical Computer Science, vol. 253, pp. 37–50. Elsevier, York, UK (2010)

7. Bagge, A.H., Haveraaen, M.: The Magnolia programming language (2013), http://magnolia-lang.org/, retrieved May 2013
8. Bagge, A.H., Haveraaen, M.: Programming by concept (2013), unpublished manuscript
9. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. Software Engineering, IEEE Transactions on 30(6), 355–371 (2004)
10. Benavides, D., Segura, S., Ruiz-Cort'es, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6), 615 – 636 (2010)
11. Bergen Language Design Laboratory: Anyxporter, https://github.com/bldl/anyxporter
12. BlackBerry: BlackBerry Developer, http://developer.blackberry.com/, retrieved May 2013
13. BlackBerry: BlackBerry 10 Native SDK 10.0.9. Software distribution (Dec 2012)
14. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. pp. 23–42. TPHOLs '09, Springer-Verlag, Berlin, Heidelberg (2009)
15. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 627–638. CCS '11 (2011)
16. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: User attention, comprehension, and behavior. In: Proceedings of the Eighth Symposium on Usable Privacy and Security. pp. 3:1–3:14. SOUPS '12 (2012)
17. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. SIGPLAN Not. 38(5), 1–11 (May 2003)
18. Hasu, T.: ContextLogger2—a tool for smartphone data gathering. Tech. Rep. 2010-1, Helsinki Institute for Information Technology HIIT, Aalto University (Aug 2010)
19. Heath, C.: Symbian OS Platform Security: Software Development Using the Symbian OS Security Architecture. Wiley (Feb 2006)
20. Hernie, D.: Windows Phone 8 security deep dive. Slide set (Oct 2012)
21. Kostiainen, K., Reshetova, E., Ekberg, J.E., Asokan, N.: Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures. In: Proceedings of the First ACM Conference on Data and Application Security and Privacy. pp. 13–24. CODASPY '11 (2011)
22. Microsoft: Microsoft Developer Network, http://msdn.microsoft.com/, retrieved Jul 2013
23. mssf-team: Mobile simplified security framework (May 2012), http://gitorious.org/meego-platform-security
24. Nokia Corporation: Nokia Developer, http://www.developer.nokia.com/, retrieved May 2013
25. Nokia Corporation: Qt Mobility 1.2: Qt Mobility project reference documentation (2011), http://doc.qt.digia.com/qtmobility/
26. Samsung: bada Developers, http://developer.bada.com, retrieved Mar 2013
27. Samsung: bada SDK 2.0.0. Software distribution (Aug 2011)
28. Tizen Project: Tizen Developers dev guide, https://developer.tizen.org/, retrieved May 2013
29. Tizen Project: Tizen SDK 2.0. Software distribution (Feb 2013)
30. Vidas, T., Christin, N., Cranor, L.: Curbing Android permission creep. In: Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011). Oakland, CA (May 2011)