Frameworks compiled from declarations: a language-independent approach

Paul Van Der Walt, Charles Consel, Emilie Balland

Frameworks compiled from declarations: a language-independent approach

Paul van der Walt Charles Consel Emilie Balland

April 19, 2016

Abstract

Programming frameworks are an accepted fixture in the object-oriented world, motivated by the need for code reuse, developer guidance, and restriction. A new trend is emerging where frameworks require domain experts to provide declarations using a domain-specific language (DSL), influencing the structure and behaviour of the resulting application. These mechanisms address concerns such as user privacy. Although many popular open platforms such as Android are based on declaration-driven frameworks, current implementations provide ad hoc and narrow solutions to concerns raised by their openness to non-certified developers. Most widely used frameworks fail to address serious privacy leaks, and provide the user with little insight into application behaviour.

To address these shortcomings, we show that declaration-driven frameworks can limit privacy leaks, as well as guide developers, independently from the underlying programming paradigm. To do so, we identify concepts that underlie declaration-driven frameworks, and apply them systematically to both an object-oriented language, Java, and a dynamic functional language, Racket. The resulting programming framework generators are used to develop a prototype mobile application, illustrating how we mitigate a common class of privacy leaks. Finally, we explore the possible design choices and propose development principles for developing domain-specific language compilers to produce frameworks, applicable across a spectrum of programming paradigms.

Keywords— Generative programming, programming frameworks, privacy controls, domain-specific languages

1 Introduction

Software reuse is agreed to be a goal in itself, for keeping applications maintainable, facilitating the development process, and avoiding repetition. To this end, software libraries have long met a need in software engineering. Going beyond libraries, programming framework popularity has been driven by advantages like ease of application development, not bothering developers with managing the life cycle of the application, and preventing deviation from an architectural style [53], while still providing access to common or shared software artefacts. Frameworks are like libraries which exercise authority: instead of a developer writing a whole application from scratch and calling routines provided by a library, frameworks manage control flow, calling snippets a developer has provided [20]. They turn full application development into a hole-filling activity, providing placeholders which may be specialised with the desired behaviour.

Frameworks are found everywhere: in the domain of mobile applications, Web programming, to gaming platforms. We see a trend emerging, where frameworks make use of domain-specific declarations as input [9, 44, 50]. These declarations dictate the structure, permissions to access resources, and behaviour of applications. Usually, a framework provider develops an external DSL (Domain-Specific Language [24]) which the framework somehow uses to influence the behaviour of the application. DSLs themselves are a well-established tool for increasing programmer efficacy as well as bridging the communication gap between application developers and domain experts [36, 42, 51, 57]. This work aims in part to explore the link between DSLs and application programming frameworks. The complexity of DSLs used to parameterise frameworks varies greatly: we find examples ranging from simple lists of permissions as in Android, through to relatively rich DSLs used to describe the structure of the application (as in QtBuilder [13]). For example, the Manifest file required by Android applications declares which resources the application may use [44]. Resources are any sources or sinks, whether real devices (*e.g.,* camera, microphone) or virtual ones (*e.g.,* address book, the Internet). Such declarations allow the framework to better answer emerging challenges such as privacy concerns, potentially giving a user insight into how their sensitive information is used. In this work we focus on these *declaration-driven frameworks* as applied to the problem of user data privacy. This is but one example where such declaration-driven frameworks are a fruitful technique: elsewhere they have been applied to Quality-of-Service (QoS) concerns [26], automatic exception management [18], or to drive application simulation before deployment [6]. We restrict our discussion to user privacy, because it is a timely and relevant problem for which the benefits of declaration-driven frameworks are clear, as we will motivate in the rest of this article.

Recently, we are seeing an explosion of new application domains, such as mo-

bile devices, using declaration-driven frameworks and the open platform model as defined by Balland, et al. [2]. When we refer to open platforms, we mean platforms with (1) public programming interfaces, giving access to (2) shared resources for applications. They include (3) a run-time environment for applications, and contribution of applications is (4) open to non-certified, third party developers. Examples include Android and iOS [35], but also the Facebook platform [21], among many others. Because it is an attractive business model to offer a platform for which third party developers can easily write applications for end users to install, this model is being widely adopted. These novel application domains pose new challenges. For example, they expose sensitive shared resources, such as the camera or contact list, to potentially untrustworthy developers. It has been shown that on Android, abuse of these resources is routine [3, 59].

Among declaration-driven frameworks, we identify a spectrum of approaches to dealing with restrictions of resource usage. Examples range from fully dynamic enforcing of permissions, as in Android, to static capability management, as in DiaSuite [9], an existing declaration-driven approach. Considering this range of approaches and concerns, our research questions are:

RQ1 What influence does expressiveness of the declaration language have on programmer guidance and permission control?

RQ2 Can concepts like resource restriction and programmer guidance be mapped into arbitrary programming paradigms?

RQ3 If not, which language features, *e.g.,* static type checking or objects and classes, are essential to enforcing privacy restrictions and providing guidance?

RQ4 How does statically *vs.* dynamically enforcing declaration semantics influence access restriction and developer guidance?

We are interested in identifying the concrete requirements resulting from various declarations, and their mapping into programming language features. To this end, we explore a case study of a declaration-driven framework hosting a potentially malicious application, inspired by DiaSuite and Android, in two widely differing programming paradigms. We demonstrate that these frameworks can mitigate a class of common privacy leaks found in Android applications [59].

Contributions

The contributions of this work are as follows.

1. Identifying concrete but programming language-independent requirements for a framework to support an open platform; these form the evaluation criteria for our case study (Sec. 2).

2. A case study for our language-independent approach (Sec. 3).

3. Providing design principles to guide future implementations of declaration-driven frameworks, comparing static *vs.* dynamic checks (Sec. 4.2).

4. Identifying the minimal requirements for a host language to support the checks and guarantees open platforms call for (Sec. 4.3).

2 Identifying the requirements

First, we identify concerns of the stakeholders in open platforms (Sec. 2.1). Technical requirements are distilled by studying existing declaration-driven frameworks. We also highlight differences in expressiveness among declaration languages. Next, we introduce our declaration language for the case study (Sec. 2.2), based on DiaSuite [9]. Finally, we instantiate the requirements for our case study (Sec. 2.3).

2.1 Requirements of open platforms

When considering open platforms, we identify the end user, the application developer, and the platform owner as stakeholders, with various concerns. By studying existing, widespread platforms, we identify emergent technical requirements which address these concerns. These requirements apply to frameworks which are to support an open platform, and will serve as evaluation criteria for frameworks developed using our proposed methodology.

[**Req1**: transparency] The user would like clarity on which shared resources will be used. *Resource declarations* should therefore specify the sources and sinks of potentially sensitive data an application uses, as well as possible side-effects. On mobile computing platforms, examples include camera or Internet access. This would allow a user to make an informed decision as to whether they trust the application enough to execute it.

[**Req2**: containment] The *data reachability* should be constrained to avoid privacy leaks [8]. Potential leaks can be predicted by determining whether a control flow path exists between components having access to various sensitive resources. This could inform the end user *how* resources are used. In Android, where this is not the case, an application may have access to both the Internet and photos, implying that

photos can be exfiltrated to an arbitrary server. The same applies to Apple's iOS: if the user gives permission to use a given resource, no information is provided regarding what the data will be used for.

[**Req3**: support] Tailored *programming support* for the developer can and should be derived from the declarations, since these provide hints towards the desired structure and behaviour of the application. For example, if a certain resource is disallowed, its API need not be available to the developer. This avoids clutter during the implementation phase.

[**Req4**: conformance] *Conformance checking*, whether static or dynamic, should be performed between the specifications and the implementation. This way a user can trust the application to conform to the declarations.

For each requirement, existing declaration languages vary widely in expressiveness, and thus in how accurately they can express the intent of specifications. At one end of the spectrum, the least expressive declaration language might cover only resource usage (**Req1**). For example, Facebook and Chrome plugins only require an application developer to specify the resources required (*e.g.,* cross-site requests, the "friend list", the user's birth date) [12,21]. Android declarations [44] go further, enforcing a certain architectural style consisting of views, called activities, and untyped communication channels between them, called intents. The declaration language in the Manifest files requires the developer to declare the components and permissions of the application. Having both resource and structural declarations potentially allows more insight into what may happen with sensitive information. However, the Android declarations are not expressive enough, since permissions apply to entire applications, not components. Based only on the declarations, a misbehaving application is indistinguishable from a reasonable one. For example, if we know an application may access the Internet and access photos, we do not know what it will send where. On the other hand, if declarations were fine-grained, per-component, (*e.g.,* Internet access only allowed for certain views) a user might determine that an application cannot exfiltrate sensitive data in the background. Unfortunately, this threat model is not speculative paranoia, but a real risk, since sensitive data is routinely exfiltrated by Android applications, most frequently via misbehaving advertisement libraries [52,59].

On the other end of the spectrum are approaches like DiaSuite [9]. Like Android, the DiaSuite declaration language imposes an architectural style. Contrary to Android, resource usage in DiaSuite is part of the architecture and specified at the component level, not globally (**Req2**). The declarations also include constructs dedicated to interactions between the components [8]. This combination allows the developer to declare how components interact with each other, and which permissions each one has. This is essential to our approach for preventing data

leaks.

Another difference is that Android does not offer application-tailored programming support. DiaSuite provides an application-tailored Java framework. It is generated from the declarations, and just like the Android framework, is not intended to be modified by the application developer. This approach allows APIs for disallowed resources to be hidden from the developer, lowering development effort (**Req3**). By contrast, Android always exposes the entire framework API, without regard to the application permissions.

iOS offers yet another model for resource usage restriction. iOS does not have declarations, but simply prompts the user if and when a sensitive resource, *e.g.,* geolocation, is about to be accessed. These checks are therefore dynamic, and their "declaration", if one may call them that, ad hoc. This gives the user the advantage of a high degree of insight into which resources are used at what moment, but the current implementation still falls short. Once a permission is granted, the application may access the sensitive resource as and when it wishes. Unsurprisingly, a common tactic among malicious applications is to wait for the user to grant a benign request, and subsequently exfiltrate data without raising suspicion [1].

Android, iOS and DiaSuite therefore all offer different forms of resource permission management, but their implementation choices influence their efficacy and usability. Android and DiaSuite verify resource usage according to declarations (**Req4**), but iOS has only *a posteriori* declarations. The dynamic checks offered by Android and iOS mean that if a developer tries to access a forbidden resource, an exception is raised. This approach risks aborting the application as a result of uncaught exceptions. This might only be discovered via testing, or worse, by end-users. Tailored programming support is also unavailable. By contrast, in DiaSuite, resource usage is enforced statically. The developer and the end-user can therefore be sure, at compile time, that all permissions required have been granted accordingly.

2.2 Core declaration language

For our case study, we use a simplified declaration language, inspired by DiaSuite [8]. The main interests of this language for our case study are that it applies to open platforms, and relies on an expressive declaration language, thoroughly illustrating the potential of declaration-driven frameworks.

2.2.1 DiaSuite

DiaSuite is a development toolkit dedicated to the *Sense/Compute/Control* (abbreviated as SCC) architectural style described by Taylor et al. [53]. This pattern

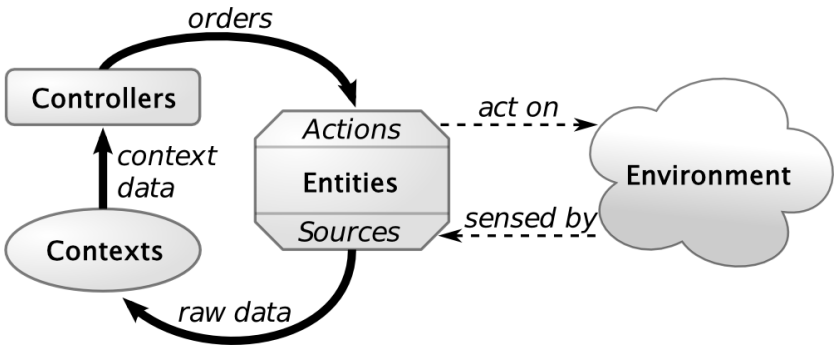

Figure 1: The *Sense/Compute/Control* paradigm, adapted from [9].

ideally fits applications that interact with an external environment. SCC applications are typical of domains such as building automation, robotics, avionics and automotive applications, but the SCC model also fits mobile device platforms.

As depicted in Fig. 1, SCC consists of three types of components: (1) *entities* correspond to managed[1] resources, whether hardware or virtual, which supply data; (2) *context components* process data; (3) *controller components* use this information to control the environment by triggering actions on entities. All components are reactive. This decomposition of applications into processing blocks on the one hand, and control flow on the other, makes data reachability explicit, and isolation more natural. When targeting a specific domain (*e.g.,* building automation or mobile phones), the platform owner defines a taxonomy of resources. On mobile devices, for example, this includes the camera, contact list, *etc.*

2.2.2 Declaration language grammar

The grammar of the declaration language associated with DiaSuite is presented in Fig. 2. It is adapted from [9], keeping only essential constructs. In this article we limit ourselves to a pragmatic presentation of the semantics of the declarations, leaving formalisation to future work.

Resources (such as camera, GPS, *etc.*) are defined and implemented by the platform, and are inherent to the application domain. Sources and actions return or accept values of a fixed type. Context and controller declarations include interaction contracts prescribing how they interact. A context can be activated by another component requesting its value (**`when required`**) or a publication of a value by another component (*i.e.,* **`when provided`** *`component`*). When activated, a context component may be allowed to pull supplementary data (denoted by the

[1]Managed resources are those which are not available to arbitrary parts of the application, in contrast to basic system calls.

```
Specification -> Declaration*
Declaration   -> Resource | Context | Controller
Type          -> Bool | Int | String | Picture | ...

Resource      -> (source srcName | action actName) as Type

Context       -> context ctxName as Type { CtxtInteract }
CtxtInteract  -> when ( required GetData?
                      | provided (srcName | ctxName)
                                 GetData? PublishSpec)
GetData       -> get (srcName | ctxName)
PublishSpec   -> (always | maybe) publish

Controller    -> controller ctrName { ContrInteract }
ContrInteract -> when provided ctxName do actName
```

Figure 2: Declaration grammar. Keywords are in bold, terminals in italic, and rules in normal font.

optional **get**) from a source or another context. Contexts which are to be pulled from must have a corresponding **when required** contract. Finally, a context might be required to publish when triggered (defined by PublishSpec). Note that **when required** contexts have no publish specification, since they are only activated by pulling, and hence return their values directly to the component which polled them. When activated, controller components can send an order, using the actuating interfaces they have access to (via **do** *actName*), for example printing text to the screen or sending an email. A visual representation can be derived from the specification and can be presented to an end-user before execution of an application, if desired.

2.2.3 Example scenario

We base our example on a well-known application which allows a user to take a picture, which is then processed by a visual filter. The picture should remain local and only be shown to the user. Since the application is distributed for free, supported by advertisement revenue, it relies on an ad component. Our threat model is that this component will try to exfiltrate the picture to a third party server. Fig. 3 shows a possible specification of this application. The camera publishes a value when the user takes a snapshot, which triggers *ProcessPicture*. On publication of the filtered image, *ComposeDisplay* is activated. Before displaying the picture to the screen, it overlays an advertisement. The ad is retrieved from the Internet by *MakeAd* and returned as a string. We assume that *IP*, *Camera* and *Screen* are provided by the platform. Note that writing the specification does not impose much overhead on the developer.

```
context ProcessPicture as Picture {
    when provided Camera
    always publish }

context ComposeDisplay as Picture {
    when provided ProcessPicture
    get MakeAd
    maybe publish }

context MakeAd as String {
    when required
    get IP }

controller Display {
    when provided ComposeDisplay
    do Screen }
```

Figure 3: The specification for our example application.

In this article we present the implementation of a declaration-driven framework, as well as the implementation of the example application using said framework. We emphasise that the intent is to provide an experience report on the implementation of the case study, and that it is provided for its illustrative value only.

2.3 Requirements, instantiated for the case study

The goal of the framework is to support the developer, and ensure certain behaviours. We now refine the requirements as identified in Sec. 2.1. We identify three concrete types of requirements: obligations, restrictions and support. Obligations are where the developer should be forced to do something, *e.g.,* implement all declared components. Restrictions are for when we want to ensure certain properties, *e.g.,* the developer may not arbitrarily access private user data. Support for the developer might include being provided with a specialised API.

[**Req1**: transparency]

- The user should be given the opportunity to approve sensitive operations.
- Restrictions: once the user approves the specification, each component should only have access to the resources explicitly granted, *e.g.,* only the `MakeAd` context should be able to query `IP`. Also, `MakeAd` should have no access to any image from the `Camera`.

[**Req2**: containment]

- Restrictions: the developer should not be able to activate components directly, except via framework methods. This control flow restriction is to enforce data reachability. Although coarse-grained, this method avoids doing expensive static analyses.

[**Req3**: support]

- Support: the publication system should be transparent to the developer. That is, the developer should merely have to write functions that return values to be published, and not have to look up which components are subscribed, *etc.* The framework must take care of the subscription and message delivery steps.

- Support: API calls for accessing resources should be made available as needed, exclusively to the components authorised to use them, based on the declarations.

- Support: all declared components require an implementation. If any are missing, the developer should be warned.

[**Req4**: conformance]

- The application should be checked to conform to the specification. If a component fails to broadcast when promised, tries to initiate unauthorised access to a resource, or otherwise deviates from the specification, the verification should fail.

Next, we present our translation of these requirements into concrete programming artefacts in the form of a framework. We also evaluate the prototypes according to these requirements.

3 From Requirements to Implementation

For our prototypes,[2] we use two radically different languages: an object-oriented, statically typed language (Java, Sec. 3.1), and a dynamic functional language (Racket, Sec. 3.2).

When considering which programming languages are widely used in mobile computing, we remark that the vast majority falls on the spectrum of statically or dynamically typed, object-oriented and/or functional languages. By choosing Java and Racket, we cover the core of languages most likely to be used. Therefore,

[2] All code is available at http://people.bordeaux.inria.fr/pwalt/code/frameworks.tgz.

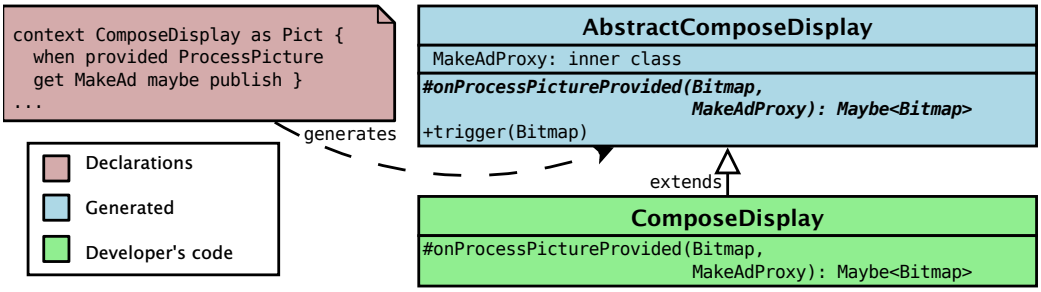

Figure 4: Schematic design of the Java prototype.

the case study should give a measure of the degree to which our methodology is language-agnostic, that is, not depending on specific programming language features from one paradigm.

The contrast between Java and Racket should substantially differentiate how the requirements are fulfilled. Although Racket offers the statically checked Typed Racket library [55], we deliberately only use dynamic features. Our reasoning is that implementation in a statically typed language is already shown to be possible by the Java version. We want to explore the design space as far away from the current implementation as possible, to clarify the impact of the paradigm on the guarantees provided.

Both implementation sections are structured as follows: a *general description* of the design of the prototype framework, the *translation* of the declarations into programming language constructs, an *example implementation* of a context, and finally an *evaluation* of the prototype with respect to the requirements.

3.1 Java prototype

The Java prototype is adapted from the system proposed by Cassou, et al. [9]. It compiles the specification into a tailored framework. Each declaration is translated into an abstract class including an abstract method with a type derived from the interaction contract of that component, which the developer must implement (see Fig. 4). Access to resources is given via specialised arguments which are passed to these generated abstract methods. The developer may regenerate the framework if the specifications change, but may never modify the generated code directly, since the implementation code remains separate from framework.

3.1.1 Translation of the declarations

We present a sketch of how the declarations are translated into Java programming artefacts. We follow the grammar given in Fig. 2 case-by-case. Declaration of a component C results in an abstract class `AbstractC`, containing one abstract

method, whose type is derived from the interaction contract. The return type of contexts is determined by the type annotation of the corresponding declaration, *e.g.,* `String` is the return type of the *`MakeAd`* context, which corresponds to Fig. 3, line 10. In the case of controllers, the return type of the abstract method is always `void`, since controllers do not compute values, but call action methods, as seen in Fig. 3, line 16.

Activation conditions The name and first parameter of the method depend on the activation conditions.
`when provided x`. The abstract method will be named `on`*`x`*`Provided`. The first argument will have the same type as *x*. For example, **`when provided`** *`ProcessPicture`* produces `onProcessPictureProvided(Bitmap p, ...)`. The `Bitmap` type results from the fact that *`ProcessPicture`* returns a picture.
`when required`. Names the abstract method `whenRequired`, without an argument. This is because activation results from a pull request, not a publication.

Data sources, actions These result in a tailored proxy passed to the method, managing access to resources.
`get x`, **`do x`**. Adds an inner class to the abstract class, containing run-time access control. An instance of this proxy is added as an argument to the `whenRequired` or `on...Provided` method, giving the developer managed access to *x*. For example, **`get`** *`MakeAd`* creates the inner class `MakeAdProxy` in *`AbstractComposeDisplay`*. Actions for controllers are handled the same way.

Publication requirements These determine the return type of the method.
`always publish`. The return type is simply the type of the context. The types in the specification language are trivially mapped to Java types, such as `Bitmap` for pictures.
`maybe publish`. The type of the context is wrapped in the option type, `Maybe<T>`.

Methods `trigger()` and `notify()` are generated to map these generically-named calls in the framework to customised names such as `onProcessPictureProvided()`.

3.1.2 Illustration with the `ComposeDisplay` context

The developer's example implementation of `ComposeDisplay` is presented in Fig. 5. We see that the developer implements the single method, `onProcessPictureProvided`, whose type corresponds to *`ComposeDisplay`* in Fig. 3.

Note that we could have avoided generation in favour of generics, for example by requiring a developer to provide a class which inherits from something like

```
1 public class ComposeDisplay extends AbstractComposeDisplay {
2   @Override protected Maybe<Bitmap>
3       onProcessPictureProvided(Bitmap p, MakeAdProxy ad) {
4         String adtxt = ad.queryMakeAdValue();
5         if(adtxt == null) {
6           return new Nothing<Bitmap>();
7         }
8         // ..do magic with image, overlay ad text..
9         return new Just<Bitmap>(p);
10      }
11    }
```

Figure 5: The implementation of the *ComposeDisplay* context.

`Context<Maybe<Bitmap>>`. However, we would lose the descriptive power of generated method names, as well as the simplicity of the resource interface.

The corresponding generated abstract class is shown in abbreviated form in Fig. 6. We hide `onProcessPictureProvided` for brevity, since it has already been discussed. The `MakeAdProxy` argument comes from the declaration **`get`** *MakeAd* (Fig. 3, line 7). Using `private` and a run-time check, we ensure that *MakeAd* is only accessible while the framework polls *ComposeDisplay*. This proxy is intended to provide access restriction, plus a simpler API. Another approach could be to pass the result of *MakeAd* by value, but our approach prevents unnecessary preemptive polling, which could pose a problem if polling has desired side effects. Note that in our approach, *MakeAd* has no access to the picture the user has taken. The `queryMakeAdValue` method gets no arguments, so the picture cannot be passed to it, even if it were *e.g.,* encoded as a `String` value.

Finally, to ensure all components are implemented exactly once, we also generate a class `AbstractRunner`, taking care of linking declared names to implementations (see Fig. 7). The abstract class defines methods like `getProcessPictureInstance()`, `getMakeAdInstance()`, *etc.* for a developer to override, and return an instance of the class implementing each component. Since `AbstractRunner` also contains the `main()` method, the developer is obliged to provide all the component bindings before being able to execute the application.

3.1.3 Evaluation of conformance to requirements

Reflecting on the requirements from Sec. 2.1, we see that our prototype conforms.

[**Req1**: transparency] This requirement is covered by the fact that the user must validate the specification before executing the application.
[**Req2**: containment] Resource access is strictly controlled by the framework, and

```
public abstract class AbstractComposeDisplay
      extends Publisher<Bitmap>
      implements Context, Subscriber<Bitmap> {

  public final void trigger(Bitmap value) {
    MakeAdProxy proxy = new MakeAdProxy();
    proxy.setAccessible(true);
    Maybe<Bitmap> v = onProcessPictureProvided(value, proxy);
    proxy.setAccessible(false);
    if(v instanceof Just) {
        notify(((Just<Bitmap>) v).just_value);
      }
    }

  protected final class MakeAdProxy {
    private MakeAdProxy() { } // prevent instantiation

    private boolean isAccessible = false;

    final private void setAccessible(boolean isAccessible) {
      this.isAccessible = isAccessible;
    }

    final public String queryMakeAdValue() {
      if (isAccessible) {
        return runner.getMakeAdInstance().requireValue();
      } else {
        throw new RuntimeException("Forbidden.");
      }
    }
  }
}
```

Figure 6: Excerpt of `AbstractComposeDisplay`.

```
public class Runner extends AbstractRunner {
  @Override
  public AbstractProcessPicture getProcessPicture() {
    return new ProcessPicture();
  }
  ...
}
```

Figure 7: Example of class binding names to implementations.

is only possible via the generated proxy classes which are given to the developer's code as function arguments. The framework polls sources and publishes values, and manages the control flow. The only way to use the framework is by calling the `main()` method, which is only available after extending `AbstractRunner`. This necessitates providing well-typed implementations of all declared components.
[**Req3**: support] For the developer, implementation is simple. The API is concise and specialised, it consists of arguments passed to the implementation, nothing else. Publication is transparent, and there is no way to omit an implementation for a component.
[**Req4**: conformance] All the properties can be checked at compile-time, except for the access to data requirements. This is checked dynamically, for each access (Fig. 6, line 25). This could potentially have been solved by using a Java extension with a more expressive ownership type system [7], but this is left as future work.

Regarding the applicability of this approach to other SDKs apart from Android, we note that we are doing no specific validation, and we make no use of specific knowledge about the Android framework, and that we treat it as a black-box interface all in plain Java. Therefore, this approach should pose no difficulties if another application environment or SDK is to be targeted. Furthermore, we note that this approach has already been successfully applied to a number of other application domains, such as structured exception handling [18], home automation [9], simulation [6], and assisted living (in the context of the HomeAssist project [14]).

Limitations. One possible attack on this system could be to use some unsupervised call, such as writing to a file with a preshared name for unauthorised communication, or performing shell executions. Importing libraries also poses a threat: singleton classes might be used for unwanted communication. In fact, libraries might allow execution of arbitrary code. However, Android demonstrates that it is feasible to sandbox and restrict system calls, and we could trivially analyse the use of `import` keywords in developer code. Particularly, we must disallow importing the reflection library, since behaviour would be very difficult to control in the presence of reflection mechanisms.

3.2 Racket prototype

In this section, we present the functional prototype. It provides the same level of support and constraint as the object-oriented prototype. Racket [23] is a descendant of Scheme, with powerful syntax transformers. It supports creating language extensions or even entire languages as libraries [56], which may have full use of the features of Racket. It also has an advanced module system [22], supporting submodules and arbitrarily many transformer stages. Finally, a library of

run-time function contracts is available. Contracts are a language extension to annotate functions with arbitrary run-time checks on input and output. An example of a contract (not to be confused with the interaction contracts of DiaSuite) is `(-> int? bool?)`, which denotes that a function must take an integer and produce a Boolean. It is worth repeating that instead of contracts, we might have used Typed Racket [55], allowing us to achieve static checks, but since implementing a framework which conforms to the requirements using a statically typed language has been done in Sec. 3.1, we choose to illustrate a dynamic solution. Note that this would not fundamentally change any guarantees, only in which phase they are checked.

3.2.1 General approach

The general design of our prototype is illustrated in Fig. 8. Our approach makes heavy use of the language extension capabilities of Racket. The framework provides a DSL for specifications, with the keywords `define-context` and `define-controller`. When evaluated, the declarations module is transformed into an application-tailored language library. This language provides the keyword `implement`, with cases for each of the declared components. For the developer this is convenient, and from the point of view of the framework it provides control. As illustrated in Fig. 8, the framework, the specifications, and the implementation are all contained in separate modules. As with the Java prototype, the developer does not modify the framework or macros. Note the use of the **`#lang`** tags: the implementation refers to the separate specification file, and the specification file refers to the framework code.

3.2.2 Translation of the declarations

Here we give a general outline of the syntax that each declaration written in a DiaSpec module will produce, and how `implement` works. Declaring a component C adds a case to the `implement` macro. Now, a developer can use the form `(implement` C f`)` to bind a lambda function f as the implementation of C. Not just any f may be provided, as the arguments to `implement` are subject to tailored function contracts. Like the Java abstract method header, the contract for f is derived from the interaction contract.

Activation conditions These define the first argument to the function f.
`when provided` ***`x`***. First argument gets type of x. For *`ComposeDisplay`*, the contract starts with `(-> bitmap%? ...)`,[3] since it is activated by *`ProcessPic-`*

[3] In reality, `bitmap%?` is shorthand for `(is-a?/c bitmap%)`, the contract builder which checks that a value is an object of type `bitmap%`.

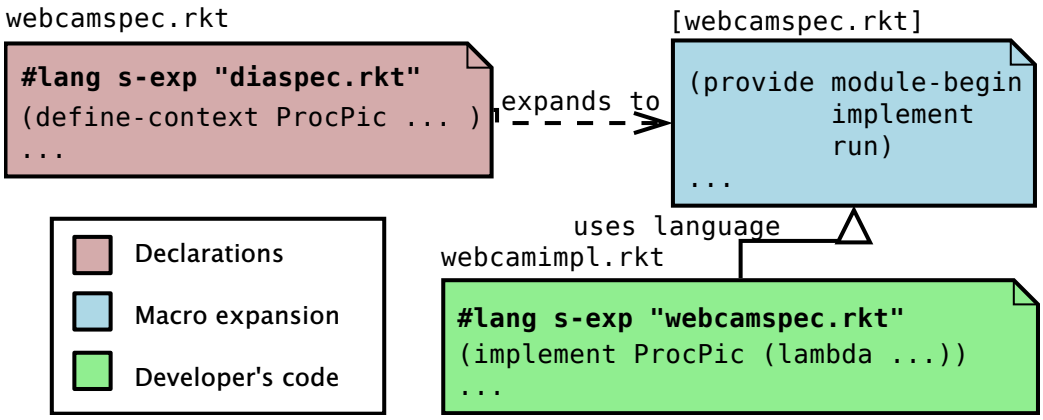

Figure 8: The Racket approach. Provided declarations are transformed into a tailored language for the implementation. The `implement` macro gets cases for each declared component.

ture publishing a picture.
`when required`. No argument added—the context was activated because another component polled its value.

Data sources, actions These determine the next argument. Comparable to the Java proxy, it is a closure providing access to the resource. This makes it convenient for a developer to query a resource, and allows the framework to enforce access control permissions. Actions for controllers are handled the same way.
`get x`. The contract of the closure becomes `(-> t?)`, where t is the output type of x. The contract so far is therefore `(-> ... (-> t?) ...)`.
`do x`. The contract of the closure becomes `(-> t? void?)` where t is the input type of x. The full contract is therefore `(-> ... (-> t? void?) void?)`. The final `void?` reflects that controllers do not return values.

Publication requirements These determine the last arguments to the function contract for contexts, corresponding to the return type. Publishing is treated using continuations, providing an equivalent to the `return` statement in Java, but hiding the unnecessary complexity of an option type (*e.g.,* `Maybe`).
`always publish`. One continuation function, meaning the final contract becomes `(-> ... (-> t? void?) none/c)`, with t the expected return type.
`maybe publish`. Two continuations to f, for publish and no-publish. The first has the contract `(-> t? void?)` with t the output type. The second continuation simply returns, for the case where the developer chooses not to publish. The complete contract is therefore `(-> ... (-> t? void?) (-> void?) none/c)`. The

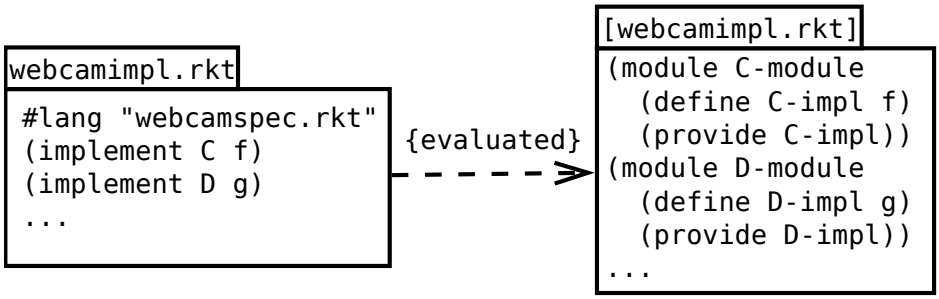

Figure 9: Separation of components using modules. The developer's code (left), and its expanded form (right). f in C cannot access D or g, because of scoping.

```
1 #lang s-exp "diaspec.rkt"
2 (define-context MakeAd String [when-required get IP])
3 (define-context ProcessPicture Picture
4   [when-provided Camera always_publish])
5 (define-context ComposeDisplay Picture
6   [when-provided ProcessPicture get MakeAd maybe_publish])
7 (define-controller Display
8   [when-provided ComposeDisplay do Screen])
```

Figure 10: Declarations of the example application, in Racket.

`none/c` contract accepts no values: it causes a run-time error if the developer does not return control to the framework via one of the provided continuations.

Furthermore, the `implement` macro wraps each f in its own submodule. As illustrated in Fig. 9, these do not have access to surrounding code, but merely export their implementations for use in the top-level implementation module. Finally, this module is statically checked to contain exactly one `(implement` C `...)` term for each component declared in the specification module.

3.2.3 Illustration with the `ComposeDisplay` context

In Fig. 10 a transliteration of the original specification, Fig. 3, is shown. Using **`#lang`** `s-exp "diaspec.rkt"` (in line 1), the developer indicates that the language is provided by `diaspec.rkt`. The syntax of the specification file is passed to the **`#%module-begin`** macro provided in `diaspec.rkt`.

A new **`#%module-begin`** macro is generated from the declarations, so that the specification module too provides a language. To illustrate how it is used, the implementation of *`ComposeDisplay`* is shown in Fig. 11. The developer points to the declaration module as the language. Implementations other than *`ComposeDisplay`* have been omitted for brevity.

```
#lang s-exp "webcamspec.rkt"
(implement ComposeDisplay
  (lambda (pic getAdTxt publish nopublish)
    (let* ([canvas  (make-bitmap pic ..)]
           [adTxt   (getAdTxt)])
      (cond [(string=? "" adTxt) (nopublish)])
      ; .. do magic, overlay adTxt on pic
      (publish canvas))))
```

Figure 11: Example implementation of `ComposeDisplay`.

```
(module webcamimpl "webcamspec.rkt"
  (module ComposeDisplay-module racket/gui
    (define/contract ComposeDisplay-impl
      (-> bitmap%? (-> string?) (-> bitmap%? void?)
          (-> void?) none/c)
      ;; this is exactly the developer's snippet
      (lambda (pic getAdTxt publish nopublish)
          ; ...
          (publish canvas)))
      ;; end snippet
    (provide ComposeDisplay-impl))
  (provide run))
```

Figure 12: The developer's code snippet is transformed into a submodule. This is the result of evaluating Fig. 11. The shaded code is simply that which the developer provided in Fig. 11.

The developer provides a lambda term for each component. In the compilation phase for the implementation module, the `#%module-begin` macro from the specifications checks that all required `implement` terms are present, *i.e.,* one per declaration. By providing the data requirement closures (*e.g.,* `getAdTxt`), we avoid giving the developer direct access to any components. The framework mediates calls to other components, so we can be sure that only legal requests are allowed. In line 5 we see the developer query the resource *MakeAd*. Next, we see how the developer obtains and uses the provided values and publishes the computed value in line 8. If the ad component returns an empty string, the developer returns without publishing (line 6). If the developer provides a conforming implementation for each component, the module can be loaded, and the complete system can be executed. Each provided `implement` term is expanded into a submodule and a contract. Fig. 12 shows the result of Fig. 11, illustrating how the code is encapsulated and how contracts are attached to the developer's implementation. After expansion, a submodule `ComposeDisplay-module` is introduced (line 2). The body of a submodule has no access to identifiers outside its lexical scope. Each implementation is bound to a name, all of which are lexically inaccessible to the developer. Access to other components is only possible via the proxy-closures the framework provides.

The provided code snippet is bound to an identifier using `define/contract` in line 4, which attaches a behavioural contract to a function. It dictates that the first argument should be a picture (`bitmap%?`), and the second argument should be a closure which evaluates to a string, `(-> string?)`. This follows from the declaration **`get`** `MakeAd`, since `MakeAd` returns strings. The third argument is a function from `bitmap%` to `void`, modelling the publication continuation. The last argument is the no-publish continuation.

The final step is to collect the implementation terms provided by the developer, allowing the framework to run the system. In the generated `#%module-begin` macro, all specified components are verified to be implemented exactly once. These checks are evaluated in the transformer phase, thus providing a static guarantee for this property. All the implementation terms such as `ComposeDisplay-impl` are collected in a convenience function called `(run)`, which the developer may call to execute the system.

3.2.4 Evaluation of conformance to requirements

Reflecting on which language mechanisms have been used to implement the various requirements, and how well they are met, we summarise as follows.

[**Req1**: transparency] As in the Java prototype, this requirement is covered because the user must validate the specification before executing the application.

[**Req2**: containment] Resource usage is controlled by the framework. The developer's code is isolated using submodules and only given access to resources via checked proxy closures.

By providing continuations as proxies, which check publications, we can be sure that the developer cannot influence the control flow. Combined with submodule scoping this ensures encapsulation of components.

[**Req3**: support] The implementation is simplified by the novel use of a tailored language extension. The developer is provided with helpful syntax errors if an implementation is omitted, and the API merely consists of the allowed resources being passed to the implementation as function arguments.

[**Req4**: conformance] The structure of the implementation is verified statically, but the types of values a developer provides are dynamically checked using contracts.

All properties resulting from the specifications are checked and enforced. We ensure the same level of restriction as the Java prototype, and have managed to give the developer a clear and concise way of implementing the components. There is no complex API to communicate with other components, and the verification is mostly static. The types of values are checked using Racket contracts, at run-time. If we switched to Typed Racket [56], the checks would be static, like in Java. Switching to Typed Racket would be trivial—it amounts to changing the language to `typed/racket` instead of `racket/gui` (in Fig. 12, on line 2), and translating the contract syntax on line 4 into the very similar Typed Racket syntax. This is a strong point of Racket: allowing us to easily use the right language for each module, gluing them together using the common Racket run-time system. We also note that it is perfectly feasible to choose different languages for different component modules. This could in principle be used to make resource access controls dynamic for certain modules, and static for others. As a concrete example where this might make sense, perhaps it would be enough to ensure at compile time that the use of the camera is authorised, but for the contact list it might make sense to check access dynamically. An application trying to iterate over all contacts to exfiltrate them would then be detected very easily. The strength of our approach is that it permits fine-grained control over where exactly in the static *vs.* dynamic design space to place permission checks for a given resource. We do not necessarily advocate always using static or dynamic checks, but rather whatever best fits the particular type of resource.

Concerning the separation into submodules to avoid unauthorised communication, it was necessary to disallow module importing and dynamic evaluation. If not, a malicious developer might escape the encapsulation. For example, importing a module M with shared mutable state into contexts A and B could provide a communication channel. This problem is dealt with the same way as described in

Java. Another potential leak would be building an expression and import modules at run-time, for example with *e.g.,* `(eval '(require ...))`. In fact, many nefarious things could be done this way. For this reason we disallow all use of **`eval`** and **`require`** in implementations. This is simply a prototype, but gives a suggestion on how to mitigate such leaks. Unfortunately by nature of the fact that Racket is a dynamic language with powerful reflection, there are probably ways to hide the binding of **`eval`**, but we consider this outside the scope of this work.

We might later consider adding a safe, or vetted, way for developers to specify which libraries they would like to use, since currently only Racket base is provided. It would be easy to provide our own `require`-like keyword. Only benign modules should be allowed to be imported, but the discussion on how to determine which qualify is out of the scope of this work. For now we assume that the platform owner decides which modules to allow, if any.

Note that it is not essential to use the language extension feature, or even Racket, to implement a similar framework, one could instead generate macros for a developer to use without imposing a DSL. Defining a language extension, however, allows full control over the implementation: we can enforce that only `implement` terms appear at the module level, or that the declarations only consist of uses of `define-context`. It also permits fine-grained control over which type of checks to apply where, in the spirit of gradual typing [49].

To summarise, we have developed a system where declarations regarding structure and behaviour of an application are used to provide a programming environment which actively ensures requirements are met. At the same time, it reduces development effort for the application developer. We do not claim that ours is the best engineering approach to implement a tailored framework, rather we argue that declaration-driven frameworks can deliver great advantages.

4 Principles

In this section we discuss the lessons learned from the implementation of the prototypes in Java and Racket. We define the design space for frameworks providing restrictions on privacy and resource usage, and give a broad outline as to the implementation choices and their trade-offs. Our aim is to guide future implementations of such frameworks.

4.1 A strong case for rich declarations

Both in object-oriented and functional settings, we see that declaration-driven frameworks potentially turn declarations from an advisory document full of promises into enforced properties. This can provide the user with valuable information on

the potential behaviour of an application. Also, from the developer's point of view, implementing an application using a tailored framework can be less laborious than using a general-purpose framework. In our example, all communication, deployment, *etc.* is taken care of by the framework. This is possible because the framework has detailed information about the structure of the application. Therefore, we believe that this new generation of frameworks can provide fundamental advantages. Even if general-purpose frameworks provide a notion of restriction, the opportunity to use available declarations to ease development is missed. Clearly, the richer the declarations used, the more the framework will be able to infer about the required shape and behaviour of the application, enabling more directed programmer guidance and more precise permission control (answering **RQ1**).

4.2 The trade-offs of static *vs.* dynamic resource restrictions

We have seen that controlling access to resources, or even more generally speaking, enforcing a certain control flow is essential to the open platform domain. For each resource the framework developer may choose to handle the restrictions either statically or dynamically, depending on the sensitivity of the resource. It is not necessary to choose one approach globally. In fact, it makes most sense to decide per-component which approach is most suitable. This is precisely the gradual typing approach as advocated by Siek, et al. [49].

As with type systems, if a static approach is chosen, an advantage is early warning if a developer performs an illegal operation, but with the cost of less accurate specifications. For example, not all requested permissions are guaranteed to be used. If a dynamic approach is chosen, an advantage is a high degree of accuracy regarding which resources are used, and when. A user can be interactively prompted to allow or deny specific requests. However, the trade-off is receiving late warnings about incorrect code and forgoing a degree of developer guidance (answering **RQ4**).

For example, as in iOS, dynamically-handled resource access controls are accurate: the user can choose to allow or disallow requests on a per-resource basis, if and when access is requested. This still would not give the user a clear view on what happens with sensitive data, though. It is possible that a legitimate request for sensitive data is used to mask exfiltration.

Compared to iOS, permissions in Android are also dynamically checked, but even less favourably. Even if a given permission is unused for a particular session it still has to be allowed by the user at install time. This is a vulnerability, since advertisement libraries routinely abuse their embedding into over-privileged applications [3, 52, 59], allowing them to exfiltrate private data the application has access to. In our approach, the permissions are also defined once for all sessions, but per-component. The developer can be helped with a specialised API per

application, and compile-time warnings about misuse of resources.

4.3 Viability of enforcing requirements is independent of programming paradigm

We have some computations which are specific to our declaration-driven approach, such as checking whether queries to resources are legal. Depending on whether we choose to process the declaration semantics statically or dynamically, we get differing support and restriction.

We observe that the choice between statically and dynamically handling the declarations is orthogonal to whether the host language is statically or dynamically typed. In fact, in general, a static type system is not even a prerequisite to being able to realise all the requirements introduced in the case study. This is evidenced by the fact that in both the Java and Racket prototypes, we implement identical guarantees.

More generally, all we need is a programming environment with at least one stage before run-time, enabling processing of the static semantics of the declarations (answering **RQ3**). Consider the Racket example, where we make no use of traditionally static features or a static type system, but implement everything using syntax transformers. Syntax transformer phases can simply be seen as extra compilation phases. Indeed, we see that a syntax transformer system generalises a static type system: a type system can be considered a limited-expressiveness programming stage. In Java, this is precisely how the properties are verified. However, since Racket is a very extensible and expressive language, it might give an optimistic impression of what can be achieved in other, less expressive, host languages. This does not invalidate our results, but means that what was easy in Racket might require more engineering in other languages.

Therefore, if we have (or can implement, whether through a declarations compiler or macros) stages, the place to handle the enforcing of requirements and obligations arising from the declarations, is in the stage(s) before run-time. In our case study, we used code generation plus the type system for the Java framework, and transformer phases for the Racket implementation, to achieve this kind of checking (answering **RQ2**). Therefore, widely varying techniques can be used to implement stages. We emphasise that in whatever programming language setting or combination of tools, one could always write an external declaration compiler (which is the pre-runtime stage we refer to)—our approach does not intrinsically rely on any specific features of Java or Racket. A strong and/or static type system is thus not required, even if static enforcing is desired.

5 Related work

Our work asks a different question than has been posed before: we attempt to take a step back and analyse the design of declaration-driven frameworks, where they have usually been engineering solutions to specific problems such as containment of sensitive data. There are a number of other approaches to this problem that should be mentioned, even if they do not aim to answer the same questions.

5.1 Rich declaration languages

Our work was inspired by DiaSuite [8, 9], and therefore most closely resembles it. However, while the articles related to DiaSuite do explain the theory of interaction contracts and the idea of a generated framework which supports the developer, the reflection on design space and requirements for implementing such a system are lacking. Furthermore, the discussion about the design of DiaSuite is exclusively in the context of Java. We therefore regard our work as an overview of the principles implicitly motivating previous work on DiaSuite, as well as a generalisation to a language-agnostic approach.

Yesod [50] is a web framework in Haskell which makes similar use of declarations to tailor the framework per application, and then to guide the developer, and statically verify the implementation to be free of broken URLs, missing components, *etc.* In the Yesod documentation, a reflection on the design space and the potential benefits of the use of declarations is similarly lacking. Therefore, it is unclear if the declarations are being put to optimal use.

5.2 TouchDevelop, a mobile application DSL

Regarding frameworks which support open platforms, we find many approaches attempting to restrict sensitive data usage and give the user more insight. For example, Xiao et al. [60] provide a domain-specific programming language based on TouchDevelop[4] [29] to facilitate static analysis, per-resource permissions, and showing a user what the potential flow of information is (*e.g.,* camera $\rightarrow$ Facebook). This is different to our approach, since a developer has to learn a new language, whereas we are able to achieve meaningful and fine-grained restrictions while allowing a programmer to use their familiar general-purpose programming language (allowing freedom to choose IDE, libraries, tools, *etc.*). Also, it would be simple to extract such "arrows" from one of our specifications.

[4]TouchDevelop is an application creation environment allowing developers to write scripts for mobile devices and publish them in an application store. It offers an imperative, statically-typed language. Xiao et al. have developed a static information flow analysis, as well as a modified run-time which allows individual resources to be replaced by anonymous values.

5.3 TaintDroid, remote real-time analysis

The authors of TaintDroid [19] propose another novel approach: real-time taint analysis run in parallel on a remote server. This approach is likely the most accurate, but incurs non-negligible costs for platform owners: effectively having to emulate all running user sessions. It illustrates the great accuracy of dynamic analysis, but we believe that a static analysis makes more sense in settings where CPU power and bandwidth are finite.

5.4 Static analysis: no developer guidance

Much other work exists, including specialised work on Android [17, 25, 27, 34, 38], looking into static analysis of existing code without modifying the platform. Mostly, it is motivated by privacy and safety concerns. These approaches have their own pros and cons, including invasive inspection of the developer's code, but especially providing no guidance at implementation time. In comparison, we require modification of the platform, but believe this is justified by gains in terms of privacy for the user and guidance for the developer. Attacking the problem using static analysis of unmodified applications also seems more difficult a problem than necessary. Our approach allows providing rules to the analysis software which should be respected upfront, as opposed to trying to extract all possibly unauthorised data flows. Furthermore, we expect that our decomposition into independent components potentially allows making assumptions on the context of each snippet of code, which would allow a less exhaustive analysis.

The work presented by Park et al. [41] should also be referenced here, as they propose a technique for parallel compilation of an application into an executable and a formally verifiable Promela/SPIN model [4]. It is not a static program analysis method per se, but our remarks regarding static analysis approaches still apply.

Other work specifically concerning static analysis for the Android platform deserves a more detailed discussion here.

5.4.1 CHEX: static vetting of Android bytecode for component hijacking vulnerabilities

The CHEX static analysis tool [33] consumes Android applications directly from the application repository, and vets them for common component hijacking vulnerabilities (this describes a class of attacks that seek to gain unauthorised access to private resources via publicly available components of vulnerable applications). An example of this class of attack is a malicious application which has no access to the contact list, using an exposed API (*Intents* or *Services*, *etc.*, in Android

terminology) of an application which does have access to the contact list, to gain unauthorised access. This works by leveraging a poorly secured application which has the privileges the malicious application aims to steal.

While an extremely useful and practical contribution, CHEX aims to detect a problem which mainly results from the Android permissions model, and is this orthogonal to the concerns we address in our work. We note that if the platform were designed using our methodology, a malicious application would not be able to interact with the application with the desired privileges, since all application and component interaction paths are statically defined and enforced by the platform run-time.

ComDroid [10] is another tool which was inspired by a similar concern. It analyses communication vulnerabilities in Android applications, allowing developers to analyse their own applications for vulnerabilities before release, or allowing market reviewers and end-users to analyse applications before installation. As with CHEX, this approach is valuable, but firmly situated in the Android ecosystem. The concerns would not generalise to another computing platform if it did not support a similar message passing interface between mutually untrusted applications. Our approach has the relative advantage that interactions are specified up-front and can be shown to the user before installation, obviating the style of static checking exhibited by CHEX and ComDroid.

5.4.2 Static user-trigger dependency analysis to detect malware

Work by Elish, et al. [16] introduces a static analysis of Android applications where calls to sensitive APIs are assigned a *TriggerMetric*. This metric is intended to capture how many of the parameters at that call site result directly from user inputs or interaction. The presence of many sensitive API calls with few static dependence relations on user triggers has been shown to be a good indication of whether a specific application should be classified as malware. This *trigger-based dependence* metric is extracted using a nontrivial Android-specific static program analysis.

In comparison with the state of the art, this is a novel approach which yields very promising results. While the approach taken in deriving the *TriggerMetric* is very different from our own, we believe that our approaches could be complementary. The static specification a developer is required to provide in our method should simplify the complexity of the static analysis to measure the *TriggerMetric*, while conversely the use of user-trigger dependence could highlight a malicious application trying to mask unwanted use of sensitive resources, even though the likelihood of this appearing using our model is decreased. This is less likely in our model since the application would have to have a legitimate reason for having a subscription relation allowing data to flow between sensitive sources and sinks so

as not to make the user immediately suspicious. An example of such a legitimate cover would be an application that is supposed to be able to send SMS messages as part of its normal operation, but also sends "premium" high-cost messages to a predefined number. In such a case, the analysis proposed by Elish, et al. would be a valuable addition to our methodology, since it could highlight the call to `sendTextMessage()` having only static arguments, as opposed to user-provided arguments.

5.5 Operating system security

Another area of research which would complement our approach is capability systems. These operating systems aim to sandbox or isolate programs with restricted permissions.

Android builds on the Linux kernel, which implements the traditional coarse-grain user-based permission model, making it difficult to follow the principle of least authority (POLA) [45]. In an attempt to achieve POLA, Android runs each application process as a separate user. However, over the years, Linux has been providing more fine-grained isolation mechanisms: this includes namespaces [5], which form the basis of so-called isolated containers, as well as mechanisms to restrict the system calls available to user processes, such as seccomp or MBOX [30, 54].

Operating systems research has been focusing on *capability-based security*, including KeyKOS [28], then EROS [47], and finally Coyotos [46]. These approaches seem superior in that they offer true separation of privilege, provided by the operating system. This way, one can be sure that child processes do not escalate permissions compared to their parent processes. Capsicum provides yet another approach to restricting system calls via capabilities [58]. For a general overview of capability-based systems, see [31].

Using a capability-based system kernel approach would offer mobile platform (and other) users a much greater degree of safety regarding the use of their private resources. Unfortunately, this would require a very profound change to the software and infrastructure already widely deployed in mobile platforms. Capability-based operating systems are not currently widespread. Our methodology on the other hand requires only minimal modifications (if any) compared to the programming language tools and run-time support libraries already deployed, thus making its adoption more feasible.

Finally, more robustness might also be achieved by incorporating into our approach an automatic exception-handling approach such as [11].

5.6 Language-level restrictions

Finally, we consider language-based approaches, which are similar to the present work. These attempt to define programming languages in such a way that it is possible to prevent access to arbitrary library code at the language level. One example is Mark Miller's *E* language [37], and the accompanying run-time library called *ELib*. As a pure-Java library, *ELib* provides inter-process capability-secure distributed programming. Its cryptographic capability protocol enables mutually suspicious Java processes to cooperate safely. Objects written in the *E* language are only able to interact with other objects according to the *ELib* semantics, enabling intra-process security with object-level granularity, including the ability to safely run untrusted library code. This technique could be used to complement our approach.

Another language-level approach is that offered by W7 [43]. The W7 approach achieves a similar goal as the *E* language, but in a slightly modified variant of the Scheme programming language. Rees shows that the primitives of Scheme suffice to support secure sharing, authentication, and more, with security properties ensured by the Scheme implementation—the 'security kernel'. For example, protection is achieved using *closures*: a procedure is not just a program, but a program coupled with its environment of origin. A procedure cannot access the environment of its caller, and the caller cannot access the procedure's environment. The caller and callee are therefore protected from one another. Sharing is accomplished through explicitly shared portions of environments, which may include procedures that allow still other objects to be shared. This allows a much finer grain of control over inter-object communication than our Racket prototype allows by default. It is worth noting that Racket also supports sandboxed evaluation, with which similar control over data sharing should be feasible.

6 Conclusions

Considering our research question, dealing with encoding constraints as concrete programming features, we have shown that strong guarantees can be built into a wide spectrum of programming paradigms. We have also demonstrated that very little is required from the target language in terms of static typing or other features. These guarantees should be possible in any language which supports pre run-time stages. If there is no such support, it can be simulated using a generative approach.

Additionally, our prototypes constitute strong evidence that declaration-driven frameworks have a lot to offer all the stakeholders in the context of open platforms. They facilitate development, increase possibilities of confining sensitive data, and give users insight into application behaviour.

While declaration-driven frameworks are widespread, to the best of our knowledge, our approach goes furthest in meaningfully exploiting declarations. Additionally, we show that functional languages can also benefit from the declaration-based approach. We hope that this work will stimulate research towards developing frameworks for open platforms which protect users' privacy.

6.1 Future work

Clearly, one of the limitations of the work presented here is the fact that we do not yet have a satisfactory approach for using external modules: we cannot be sure that the privacy restrictions remain enforced unless we block libraries that are not explicitly whitelisted, which does not scale well. The problem would be solved if we could require components to be pure. An implementation in Haskell, perhaps using the Safe Haskell extension, is the first possibility that comes to mind. This will be our next project.

Another promising avenue to explore would be to generate, from the declarations as proposed here, rules to be verified by a static analysis tool. This way, after the developer implements the components but before a user installs the application on their device, an analysis could be performed to ensure that private data does indeed remain contained.

For example, an application implemented using the Java approach might be analysed using one of the available static analyses which extracts explicit data flow [32]. Alternatively, if performance is not an issue, a real-time data flow tracking approach such as the one proposed by Nair et al. [40]. In either case, this output could be compared to rules which could easily be extracted from the declarations. For example, we would check that the camera source never reaches the advertisement component or the IP sink. This way, we could further leverage the information a rich declaration language provides us. This information could also be used to present the possibly problematic data flows to the end user in an understandable graphical format, which would dramatically increase insight into program behaviour for the end user. Exactly how to present this information to a non-expert user in an understandable fashion is another avenue of research. Another promising project is AppGuarden [39]. It proposes a tool called EviCheck, which can statically analyse an Android application for conformance to data flow rules, but to date little information is given. It seems very promising, however, to try to generate such data flow rules from our declarations and have them analysed by EviCheck.

In a dynamic setting, as in Racket, the question is more complicated. Higher-order dynamic functional languages such as Racket do not trivially expose a control flow graph [48]. However, recently there have been efforts towards static analyses of such languages [15]. Unfortunately, we do not know of approaches that are

feasible for practical use.

We remark that undesired privacy leaks usually happen through assignments to variables in imported libraries or file system I/O, *etc.* Therefore, it would be useful to verify that context components are pure (*i.e.,* side-effect free). If such a purity analysis is not offered by the host system (in Java or Racket, as opposed to Haskell, for example), it would be interesting to isolate each component in an OSGi bundle, or a separate Java VM. Operating system research towards capability-based security systems [28, 46, 47, 58] demonstrates that resource usage and system call restriction is possible at the application level. We suggest placing similar control at the level of individual components should be feasible. This should make it easier to intercept such potentially dangerous system operations as file I/O at the component level, obviating the need for pure components.

Another logical step in this work would be to do an implementation effort analysis. Currently, the presented prototypes are only informally evaluated in this regard, and a full analysis is outside the scope of this work. We claim that the implementation effort was comparable, but before this methodology could be adopted on a wider scale it would be necessary to study this in-depth. This could be done by recruiting various programmers from industry as test subjects and having them implement framework generators in various programming languages, while measuring the number of hours required and tallying the number of implemented features, for example.

Finally, investigation should be done regarding distribution of binaries. Our approach is intended to apply to the scenario where developers submit their applications to a repository controlled by the platform owner, whence they can be downloaded by end users. In principle, submission of the specifications in non-compiled form, along with the implementation in either source or compiled form, should suffice. The application store could then compile the declarations, and if the supplied implementation binaries work with the independently compiled framework, the guarantees we provide should hold. In this situation, the end user must trust the platform owner to compile the application in a way that does not weaken the guarantees given by the specifications.

Acknowledgements The authors would like to thank Damien Cassou for their constructive criticism and proofreading of early drafts. We also thank the anonymous reviewers for their effort. Their constructive and actionable comments have significantly improved this paper. The first author was jointly funded by the Conseil Régional d'Aquitaine, grant 6932-CRA-2012, and an INRIA CORDI-S grant.

References

[1] Yuvraj Agarwal and Malcolm Hall. ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 97–110, New York, NY, USA, 2013. ACM.

[2] Emilie Balland and Charles Consel. Open platforms: New challenges for software engineering. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, pages 3:1–3:4, New York, NY, USA, 2010. ACM.

[3] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to Android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, New York, NY, USA, 2012. ACM.

[4] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer Science & Business Media, 1st edition, 2008.

[5] Eric W. Biederman. Multiple Instances of the Global Linux Namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112, Ottowa, Ontario, Canada, July 2006.

[6] Julien Bruneau and Charles Consel. Diasim: a simulator for pervasive computing applications. *Software: Practice and Experience*, 43(8):885–909, 2013.

[7] Nicholas Cameron and James Noble. Encoding ownership types in Java. In *Objects, Models, Components, Patterns*, pages 271–290. Springer, 2010.

[8] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. Leveraging software architectures to guide and verify the development of Sense/-Compute/Control applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 431–440, New York, NY, USA, 2011. ACM.

[9] Damien Cassou, Julien Bruneau, Charles Consel, and Emilie Balland. Toward a tool-based development methodology for pervasive computing applications. *IEEE Transactions on Software Engineering*, 38(6):1445–1463, 2012.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[11] Kwanghoon Choi and Byeong-Mo Chang. A lightweight approach to component-level exception mechanism for robust Android apps. *Computer Languages, Systems & Structures*, 44, Part C:283–298, 2015.

[12] Chrome developers. Developing Chrome extensions: Declare permissions. https://developer.chrome.com/extensions/declare_permissions, 2015. Accessed: February 2015.

[13] Matthias Kalle Dalheimer. *Programming with QT: Writing portable GUI applications on Unix and Win32*. O'Reilly Media, 2010.

[14] Lucile Dupuy, Hélène Sauzéon, and Charles Consel. Perceived needs for assistive technologies in older adults and their caregivers. In *womENcourage 2015*, Uppsala, Sweden, September 2015.

[15] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 177–188, New York, NY, USA, 2012. ACM.

[16] Karim O. Elish, Xiaokui Shu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Xuxian Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49:255–273, 2015.

[17] Karim O. Elish, Danfeng Daphne Yao, Barbara G. Ryder, and Xuxian Jiang. A static assurance analysis of Android applications. *Virginia Polytechnic Institute and State University, Tech. Rep*, 2013.

[18] Quentin Enard. *Development of dependable applications: a design-driven approach*. PhD thesis, Université Sciences et Technologies–Bordeaux I, May 2013.

[19] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[20] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[21] Jesse Feiler. *How to Do Everything: Facebook Applications*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 2008.

[22] Matthew Flatt. Submodules in Racket: You want it when, again? In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 13–22, New York, NY, USA, 2013. ACM.

[23] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. http://racket-lang.org/tr1/, Version 6.2.1.

[24] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[25] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, May 2013.

[26] Stéphanie Gatti. *A step-wise approach for integrating QoS throughout software development process*. PhD thesis, Université de Bordeaux, February 2014.

[27] Clint Gibler, Jonathan Crussel, Jeremy Erickson, and Hao Chen. AndroidLeaks: Detecting privacy leaks in Android applications. Technical report, UC Davis, 2011.

[28] Norman Hardy. KeyKOS Architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, October 1985.

[29] R. Nigel Horspool and Nikolai Tillmann. *TouchDevelop: Programming on the Go*. The Expert's Voice. Apress, 3rd edition, 2013. available at https://www.touchdevelop.com/docs/book.

[30] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX Annual Technical Conference*, pages 139–144, San Jose, CA, 2013. USENIX.

[31] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[32] Yin Liu and Ana Milanova. Static analysis for inference of explicit information flow. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 50–56, New York, NY, USA, 2008. ACM.

[33] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[34] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in Android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[35] Dave Mark and Jeff LaMarche. *Beginning iPhone Development: Exploring the iPhone SDK*. Apress, 2009.

[36] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[37] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[38] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android apps through symbolic execution. *SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

[39] Mobility + Security Group, University of Edinburgh. AppGuarden. `http://groups.inf.ed.ac.uk/security/appguarden/Home.html`, 2015. Accessed: February 2015.

[40] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008. Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007).

[41] Heejong Park, Avinash Malik, and Zoran Salcic. Compiling and verifying SC-SystemJ programs for safety-critical reactive systems. *Computer Languages, Systems & Structures*, 44, Part C:251–282, 2015.

[42] Eric S. Raymond. *The Art of UNIX Programming*, chapter 8. Pearson Education, 2003.

[43] Jonathan Allen Rees. *A security kernel based on the lambda-calculus*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[44] Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly, Beijing, China, 2009.

[45] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep 1975.

[46] Jonathan S. Shapiro, Michael S. Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In G. Klein, editor, *Proceedings of the NICTA Formal Methods Workshop on Operating Systems Verification*, NICTA Technical Report 0401005T-1, Sydney, Australia, 2004. National ICT Australia.

[47] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[48] Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[49] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[50] Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly, 2012.

[51] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[52] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in Android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, 2012.

[53] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[54] The Linux Kernel Developers. SECure COMPuting with filters. Online, `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`, 2015. Accessed: August 2015.

[55] Sam Tobin-Hochstadt and Matthew Flatt. Advanced macrology and the implementation of Typed Scheme. In *Proceedings of the 8th Workshop on Scheme and Functional Programming*, pages 1–14. ACM Press, 2007.

[56] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[57] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[58] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *Proceedings of the USENIX Security Symposium*. USENIX, 2010.

[59] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40, New York, NY, USA, 2012. ACM.

[60] Xusheng Xiao, Nikolai Tillmann, Manuel Fähndrich, Jonathan de Halleux, and Michal Moskal. User-aware privacy control via extended static information-flow analysis. In Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors, *ASE*, pages 80–89. ACM, 2012.