Declaration-Driven Frameworks: A Language-Agnostic Approach

Paul van der Walt, Charles Consel, Emilie Balland
first.lastname@inria.fr
INRIA Sud-Ouest & University of Bordeaux, France

***Abstract*—Programming frameworks are an accepted fixture in the object-oriented world, motivated by the need for code reuse, developer guidance, and restriction. Notably, open platforms, offering "app stores" to non-certified developers, rely on frameworks. A new trend is emerging where frameworks supporting open platforms utilise domain-specific declarations to address concerns such as privacy. These declarations drive the structure and behaviour of the resulting application. Although many popular platforms such as Android are based on declaration-driven frameworks, their current implementations provide ad hoc and narrow solutions to concerns raised by their openness. In particular, most widely used frameworks are limited to a single programming paradigm and domain, and ignore serious privacy leaks.**

To address these shortcomings, we show that declaration-driven frameworks can provide privacy guarantees and guide developers in a wide spectrum of programming paradigms. To do so, we identify concepts that underlie declaration-driven frameworks. We apply them uniformly to both an object-oriented language, Java, and a dynamically typed functional language, Racket. The resulting programming frameworks are used to develop a realistic prototype mobile application, illustrating how we mitigate a common class of privacy attacks. Finally, we propose principles for developing declaration-driven frameworks applicable across a spectrum of programming paradigms.

I. Introduction

Software reuse is agreed to be a goal in itself, for keeping applications maintainable, facilitating the development process, and avoiding repetition [1]. To this end, software libraries have long met a need in software engineering. Concerns like ease of development, not bothering developers with managing an application's life cycle, and preventing deviation from the architectural style, have driven the development of programming frameworks. Frameworks are like control-freak libraries: instead of a developer writing a whole application from scratch and calling routines provided by a library, a framework takes over and manages the control flow, calling the snippets a developer has provided.

Their advantages typically include (1) reducing development effort by guiding the developer, (2) restricting to a particular architectural style [2], and (3) fulfilling the needs previously met by libraries, *e.g.,* providing easy access to common or shared software artefacts.

More precisely, programming frameworks are defined by Fayad *et al.* [3] as a software engineering technique employing *inversion of control*[1], for creating applications through extension. Contrary to the use of libraries, they can be seen as a technique to turn full app development into a hole-filling activity. The framework provides placeholders which may be filled in with the desired behaviour.

Frameworks are found everywhere from smartphone, to Web, to gaming platforms. We see a trend emerging, where frameworks make use of domain-specific declarations as input [4]–[6]. These declarations dictate the structure, resource permissions, and behaviour of applications. For example, the Android manifest file declares an app's permission to use resources [4]. We call resources any sources or sinks, whether real devices (*e.g.,* cameras, microphones) or virtual ones (*e.g.,* address book, the Internet). Such declarations allow the framework to better answer emerging challenges such as privacy concerns, providing support to the developer, and giving a user insight into how their sensitive information is used. In this work we will focus on such *declaration-driven frameworks*.

Recently, we are seeing an explosion of new application domains, such as smartphones, using the declaration-driven framework and open platform model. 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 apps is (4) open to non-certified, 3rd party developers. Examples include Android and iOS [7], but also the Facebook platform [8], among many others. Because it is an attractive business model to offer a platform for which 3rd party developers can easily write applications for end users to install, the open platform model is being widely adopted.

These novel application domains pose new challenges. For example, they expose sensitive shared resources, such as the camera or contacts, to 3rd party developers. It has been shown that in Android, routine abuse of these resources is widespread [9], [10]. Among declaration-driven frameworks, we identify a spectrum of approaches to dealing with restrictions of resource usage. Examples range from fully dynamic, as in Android, to static, as in DiaSuite [6], an existing declaration-driven approach. Finally, to encourage adoption,

[1]Inversion of control is known informally as the Hollywood Principle: "don't call us, we'll call you". In software engineering, this refers to the situation where the life cycle of the application is not the responsibility of the application developer – they just provide various components which are then called as required by the run-time system.

platform providers want to facilitate the development process as much as possible. Our research questions are therefore:

- What influence do various declarations have on restriction and programmer guidance?
- Can concepts like restriction and programmer guidance be mapped into programming language constructs, for widely different paradigms?
- Does the target paradigm impact the viability of checks and guarantees?
- How does static *vs.* dynamic declaration processing influence restriction and guidance?
- Which language features, such as static type checking or language extension, are essential to enforcing restrictions and providing guidance?

We are interested in identifying the requirements resulting from the declarations and in their mapping into programming language features. We will explore a concrete case study inspired by Android. To make the case study as meaningful as possible, it is implemented in two widely differing programming paradigms. We demonstrate that our prototype frameworks mitigate a class of common privacy leaks in Android apps [9].

Contributions: Our contributions are (1) identifying the language-agnostic minimal requirements to be able to implement the checks and guarantees open platforms call for (Sec. II), (2) showing how open platform requirements map into a wide spectrum of programming languages, *e.g.,* statically or dynamically typed, object oriented or functional, (3) implementing such a framework in two very different programming languages (Sec. III) and finally, (4) synthesising general principles from our case study to guide future implementations in a potentially wide spectrum of programming languages (Sec. IV).

II. Declaration-driven programming frameworks

First, we identify requirements for the stakeholders in open platforms (Sec. II-A). We will validate these requirements by comparing them to existing declaration-driven frameworks. We also highlight varying expressiveness among declaration languages. Next, we introduce our declaration language (Sec. II-B), based on DiaSuite [6]. We use DiaSuite as a starting point because it relies on an expressive declaration language, allowing us to illustrate the potential of declaration-driven frameworks. Finally, we instantiate the requirements for our prototype framework (Sec. II-C). To illustrate our approach in a spectrum of programming paradigms, we implement the case study in two very different languages (Sec. III).

A. Requirements of open platforms

When considering declaration-driven frameworks in open platforms, we observe that the different stakeholders have various concerns:

- The end user would like clear insight into resource usage (*e.g.,* camera, address book) by 3rd party applications: *which* resources are requested, *how* they are used.
- The platform provider wants to facilitate the development process as much as possible, to encourage adoption of the platform.
- The 3rd party developer is interested in high-level programming support, and abstraction from platform details, leading to greater portability, *e.g.,* hardware agnostic implementations.

Given these concerns, we suggest more precise requirements. They are not arbitrary: by comparing them to the declaration languages provided by prevalent and successful frameworks, we validate that these are emergent requirements of real-world stakeholders. We will show that for each requirement, declaration languages vary widely in expressiveness and precision.

[**Req1**] 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 a given application uses, as well as possible side-effects. On Android, examples include camera or Internet access. This allows a user to make an informed decision on whether they trust the application enough to execute it.
[**Req2**] The *data reachability* should be constrained to maintain the safety of sensitive information [11]. Potential leaks might be predicted, by clarifying whether a data flow path exists between components, which have access to various sensitive resources. In Android, for example, the application may have access to both the Internet and photos, implying that photos can be exfiltrated to an arbitrary server.
[**Req3**] Tailored *programming support* for the developer can and should be derived from the declarations, since they provide hints towards the app's desired structure or behaviour. For example, if the declarations prohibit a certain resource from being used, its API should not be available to the developer. This avoids confusion and clutter during implementation.
[**Req4**] Static *verification* should be performed. If a developer can compile the implementation, it should be guaranteed to conform to the declarations. This way a user can trust the declarations to be meaningful for the application.

At one end of the spectrum, the least expressive declaration language might cover only resource usage requirements (**Req1**). For example, Facebook only requires an app developer to specify the resources required (*e.g.,* cross-site requests, the "friend list", the user's birth date) [8]. Android declarations [4] go further, enforcing a certain architectural style consisting of different views, called activities, and untyped communication channels between them, called intents. The interactions between these components are controlled by the framework, *i.e.,* the system handles delivery of intents by calling the appropriate activities. The underlying declaration language, used in the Manifest files, forces the developer to declare the application's components and permissions. Having resource declarations in combination with 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 are fine-grained (*e.g.,* Internet access only allowed for certain views) a user might see that an application cannot exfiltrate sensitive data in the background. Unfortunately, this is not speculative paranoia, but a real risk, since sensitive data is routinely exfiltrated via real-world Android applications, often via misbehaving advertisement libraries [9].

On the other end of the spectrum, we have approaches with expressive declarations, like DiaSuite [6]. Like Android, DiaSuite's declaration language imposes an architectural style, *Sense/Compute/Control* (explained in Sec. II-B1). Contrary to Android, DiaSuite's resource usage is part of the architecture and specified at the component level, not globally (**Req2**). The declarations also include constructs dedicated to the interactions between the components [11]. This combination allows the developer to declare how components may interact with each other, and which permissions each one has. This kind of data and control flow restriction is essential to our approach for containing sensitive data.

Another difference is that Android does not offer application-tailored programming support. DiaSuite provides a customised Java framework, generated from the declarations. In doing so, APIs for disallowed resources are made unavailable to the developer. This strategy lowers development effort, since only essential API calls are available (**Req3**).

iOS[2], Android and DiaSuite all offer resource permission management. Android and DiaSuite verify resource usage according to the declarations (**Req4**), but all three differ in how they enforce the declared permissions. In Android, if a developer tries to access a forbidden resource, a run-time exception is raised. This approach risks crashing as a result of uncaught exceptions. This might only be discovered via testing, or worse, by end-users. By contrast, in DiaSuite, resource usage is checked statically. The developer and the end-user can therefore be sure that all permissions required have been granted accordingly. iOS on the other hand simply prompts the user if and when a sensitive resource will be accessed for the first time, making resource declarations dynamic and on-demand. This has the drawback that from then on, the application may access the sensitive resource as and when it wishes. Unsurprisingly, a common tactic among malicious apps is to wait for the user to grant a benign request, subsequently exfiltrating data without raising suspicion.

B. Core declaration language

For our case study, we will use a simplified declaration language, modelled on DiaSuite [11]. The main interests of this language for our case study are that (1) it applies to open platforms, for which declaration-driven frameworks have shown to be an attractive development approach; (2) it relies on an expressive declaration language, raising the bar for the implementation and thoroughly illustrating the potential of declaration-driven frameworks.

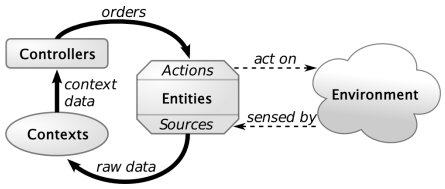

Fig. 1. The *Sense/Compute/Control* paradigm.

```
Declaration   -> Resource | Context | Controller
Type          -> Bool | Int | String | ...
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 | nothing)
PublishSpec   -> (always | maybe) publish
Controller    -> controller ctrName { ContrInteract }
ContrInteract -> when provided ctxName do actName
```

Fig. 2. Declarations grammar. Keywords are in bold, terminals in italic, and rules in normal font.

1) DiaSuite: DiaSuite is a development methodology which is dedicated to the *Sense/Compute/Control* architectural style described by Taylor *et al.* [2]. This pattern ideally fits apps that interact with an external environment. SCC apps are typical of domains such as building automation, robotics, avionics and automotive applications, but we will see that it also fits smartphone apps.

As depicted in Fig. 1, this architectural pattern consists of three types of components: (1) *entities* correspond to managed[3] resources, whether hardware or virtual, supplying data; (2) *context components* process (filter, aggregate and interpret) data; (3) *controller components* use this information to control the environment by triggering actions on entities. Furthermore, all components are reactive. This decomposition of apps into processing blocks and data flow 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 for applications in this domain. On smartphones, for example, this includes the camera, phone book, Internet, *etc.*.

2) Experimental declaration language: In this case study, we will use the minimal specification language shown in Fig. 2.

A specification is a list of `Declarations`. Resources are defined and implemented by the platform owner. We will see their use in the example scenario. Context and controller declarations include interaction contracts [11], which prescribe how they interact. A context can be triggered by either a pull request (`when required`) or a publication from another component (*i.e.,* `when provided` *component*). When activated, a

[2]iOS does not use declarations, but since it does have a crude resource access control mechanism, we consider its alternative approach in the spectrum of possibilities.

[3]Managed resources are those which are not available to arbitrary parts of the application, in contrast to basic system calls such as querying the current date.

```
context ProcessPic as Picture
  { when provided Camera always publish }
context AssemblePic as Picture
  { when provided ProcessPic get MakeAd maybe publish }
context MakeAd as String { when required get IP }
controller ScreenController
  { when provided AssemblePic do Screen }
```

Fig. 3. The specification for our example application.

context component may need to pull data (denoted by the optional `get`). Note that corresponding contexts must have a `when required` contract. Finally, a context might be required to publish new values (defined by `PublishSpec`). Note that `when required` contexts never publish, since they are only activated by pull. When activated, controller components can send orders, using the actuating interfaces of components they have access to (*i.e.,* `do id`), for example to print text to the screen.

3) Example scenario: Our running example comes from the domain of smartphone applications. We base our example on a well-known app which allows a user to take a picture, which is then instantly processed by a visual filter. Since the app is free, it is supported by advertisement revenue, thus relies on an ad library, which will try to steal the picture.

Fig. 3 shows a possible specification of this app using our language. The camera publishes a value when the user takes a snapshot, which triggers the context that processes the picture, *ProcessPic*. When it publishes, *AssemblePic* is activated. Before displaying the picture to the screen, *AssemblePic* overlays an advertisement, which is downloaded from the Internet (we assume that a 3rd party is responsible for this advertisement "library"). Since something might go wrong with the ad library, *AssemblePic* does not have to publish. Note that writing the specification does not impose much overhead on the developer.

- *MakeAd* downloads a string representing an advert when queried: it needs access to the Internet (the *IP* device). We assume it is implemented in the ad library.
- *ProcessPic* should take a raw image (requiring `Camera` permissions) and return a `Picture`.
- *AssemblePic* combines the picture and ad, ready for display via the *ScreenController*. The controller needs access to the platform-provided *Screen* device. It may publish a result (`maybe publish`).

These declarations should be translated into restrictions for the developer, to be enforced by the framework.

C. Requirements, instantiated for the case study

The goal of the framework is to support the developer, and ensure certain behaviours. We now revisit these requirements, as identified in Sec. II-A. 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 should not arbitrarily access private user data. Support is when the developer should be guided, *e.g.,* by being provided with a specialised API.

Let us instantiate each of the requirements for our case study.

[**Req1**]

- Restriction: each component should only have access to the resources explicitly granted. For example, only the *MakeAd* context should be able to query the *IP* device. Also, *MakeAd* should not have access to any `Picture`.

[**Req2**]

- Restrictions: the developer should not be able to activate components by arbitrarily broadcasting or polling components, except via framework methods.
- Support: the publication system should be transparent to the developer.

[**Req3**]

- Support: API calls for accessing resources should be made available as needed, exclusively to the components authorised to use them, based on the declarations. For example, *AssemblePic* should have *MakeAd*'s API in scope, easily accessible.
- Obligation: all declared components require an implementation.

[**Req4**]

- Restrictions: if a developer fails to broadcast when promised, tries to access illegal resources, or otherwise deviates from the specification, the compiler should halt.
- Support: early warning if the app does not conform – it does not compile.

Next, we will show our translation of these requirements into concrete programming artefacts in the form of a framework.

III. From Requirements to Implementation

For our prototypes[4], we use two radically different languages: an object-oriented, statically typed language (Java), and a dynamically typed functional language (Racket [12]). The contrast between these languages should substantially differentiate the implementations and their non-functional properties, *e.g.,* static *vs.* dynamic privacy protection.

In Sec. III-A, we deal with the Java prototype, in Sec. III-B we present the Racket prototype. Both are structured as follows: a *general description* of the design of the prototype framework, the *translation* of the declarations into programming language constructs, an *illustration* of an implemented context, and finally an *evaluation* of the prototype with respect to the requirements.

A. Java prototype

1) General design: For the Java prototype, we generate a tailored framework from the specifications. Each component declaration is translated into an abstract class, which the

[4]All our code is available online: `http://phoenix.inria.fr/research-projects/diasuite`. The download includes a prototype Android app illustrating data theft. It is not discussed in this paper.

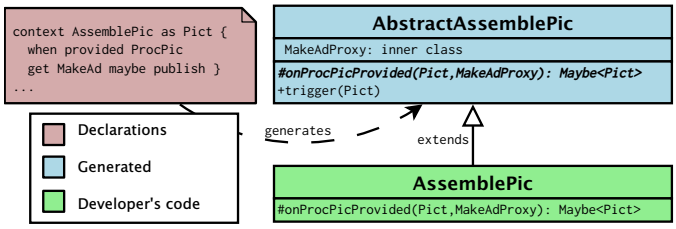

Fig. 4. Schematic design of the Java prototype.

```
1  public class AssemblePic extends AbstractAssemblePic {
2    @Override protected Maybe<Bitmap>
3        onProcessPicProvided(Bitmap p, MakeAdProxy ad) {
4          String adtxt = ad.getAdText();
5          if(adtxt == null) return new Nothing<Bitmap>();
6          // ..do magic with image, overlay ad text..
7          return new Just<Bitmap>(p); }}
```

Fig. 5. The implementation of *AssemblePic*.

developer must extend (see Fig. 4). We generate an abstract method with a type derived from the declaration, which the developer will have to implement. Access to resources is only given via specialised arguments which are passed to this method.

2) Translation of the declarations: This is a sketch of how to translate various constructs into Java programming artefacts. We follow the grammar given in Fig. 2.

Introduction of a component C results in an abstract class `AbstractC`, with an abstract method, its type derived from the interaction contract. In the case of controllers, the return type of the abstract method is always `void`, since these do not compute values.

a) Activation conditions: The method's naming and first parameter depend on its activation condition.

`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 ProcessPic` produces `onProcessPicProvided(Bitmap p, ...)`. The `Bitmap` type results from the fact that *`ProcessPic`* returns a picture.

`when required`. Names the abstract method `whenRequired`, without an argument. This is because they are not activated as a result of a publication of a value, but a pull request.

b) Data requirements: 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. An instance of this proxy is passed to the `whenRequired` or `on..Provided` method, giving the developer managed access to x. For example, `get IP` creates the inner class `IPProxy` (to be explained shortly). Controllers' actions are handled the same way.

`get nothing`. No proxy is created.

c) Publication requirements: These determine the return type of the method.

`always publish`. The return type is simply the context's type. The types in the specification language are mapped to Java types, such as `Bitmap`.

`maybe publish`. The context's type is wrapped in an option type, *i.e.,* `Maybe<T>`.

3) Illustration with the `AssemblePic` context: The developer's code is presented in Fig. 5. This context superimposes downloaded advert text onto the captured image, and publishes the composed image if successful. The advert is pulled off the Internet by the `MakeAd` context, which simulates an ad library.

Here we see that the developer implements the single method, `onProcessPicProvided`, whose type corresponds to `AssemblePic` in Fig. 3.

```
1  abstract class AbstractAssemblePic extends Publisher<Bitmap>
2        implements Context, Subscriber<Bitmap> { ...
3    public void trigger(Bitmap value) {
4      MakeAdProxy proxy = new MakeAdProxy();
5      proxy.setAccessible(true);
6      Maybe<Bitmap> v = onProcessPicProvided(value, proxy);
7      proxy.setAccessible(false);
8      if(v instanceof Just)
9          notify(((Just<Bitmap>) v).just_value); }
10   protected final class MakeAdProxy {
11     private MakeAdProxy() { }
12     private boolean isAccessible = false;
13     private void setAccessible(boolean isAccessible) {
14       this.isAccessible = isAccessible; }
15     public String getAdText() {
16       if (isAccessible)
17         return runner.getMakeAd().onRequiredMakeAd();
18       else
19         throw new RuntimeException("Forbidden."); }}}
```

Fig. 6. Excerpt of `AbstractAssemblePic` including the `MakeAd` proxy.

Note that we could have avoided generation in favour of generics, for example by requiring a developer to provide a class which inherits from `Context<Maybe<Bitmap>>`. However, we would lose the descriptive power of generated method names, and the simplicity of the resource interface.

Now, we will look at the mechanisms supporting the developer. The generated abstract class is shown in abbreviated form in Fig. 6. We hide `onProcessPicProvided` for brevity, since it has already been discussed. The `MakeAdProxy` argument comes from the declaration `get` *`MakeAd`* (Fig. 3, line 4). Using `private` and a run-time check, we ensure that *`MakeAd`* is only accessible while the framework polls *`AssemblePic`*. This proxy is intended to provide access restriction, plus a simpler API. A simpler approach might be passing `MakeAd`'s result by value, but our approach prevents unnecessary preemptive polling. Note that in our approach, *`MakeAd`* has no access to the picture the user has taken.

The `trigger()` and `notify()` methods are generated to map the generic calls in the framework to customised names such as `onProcessPicProvided()`. This way we can use a generic layer of code to execute any scenario, only generating a small amount of glue code which dispatches the generic calls.

Finally, to ensure all components are implemented exactly once, we also generate a class `AbstractRunner`, taking care of linking declared names to implementations. Its usage is shown in Fig. 7. The abstract class defines methods like `getProcessPicInstance()`, `getMakeAdInstance()`, *etc.* for a developer

```
1 public class Runner extends AbstractRunner {
2     @Override public AbstractProcessPic getProcessPic() {
3         return new ProcessPic(); } ... }
```

Fig. 7. Example implementation's deployment/binding description.

to override, returning an instance of the class implementing each component. Since `AbstractRunner` also contains the framework's `main()` method, the developer must provide all the component bindings before being able to run the app.

4) Evaluation of the prototype: We might improve our prototype by defining custom Java annotations, *e.g.,* `@ProcessPic`. These could be used to mark which class is the implementation of which declared component. This approach could generate above-mentioned abstract classes as needed, and the `AbstractRunner` class would become unnecessary. We consider this an optional improvement to what we have presented here, but functionally it would be identical. We already check most properties statically, and this would not improve using annotations. Reflecting on the requirements from Sec. II-A, we see that our prototype conforms.

[**Req1**] Resource access is strictly controlled by the framework, and is only possible via the generated proxy classes which are given to the developer's code as function arguments.
[**Req2**] 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**] 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 a component's implementation.
[**Req4**] 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 16). This could alternatively have been solved by using a Java extension with a more expressive ownership type system [13], but we decided remain with native Java.

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. However, Android demonstrates that it is feasible to restrict system calls – in our case they should only be available to resources, and not developer-provided components.

B. Racket prototype

We will now look at the functional prototype. We provide the same level of support and constraint as in the previous prototype. We start with a general description of Racket.

1) About Racket: Racket [12] is a descendant of Scheme. It has powerful syntax transformers (a.k.a. macros [14]). Racket supports creating language extensions or even entire languages as libraries [15]. These new languages may have full use of Racket's features. It also has an advanced module system [16],

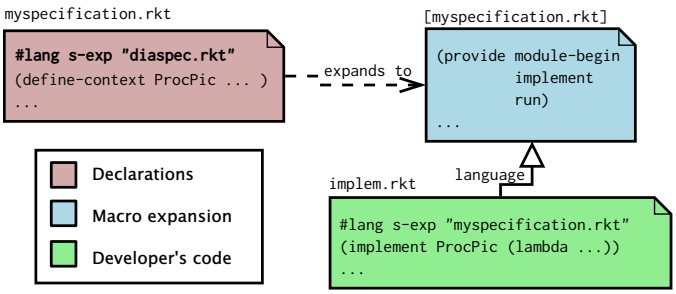

Fig. 8. The Racket approach. Provided declarations are transformed into a tailored language for the implementation. The `implement` transformer gets cases for each declared component.

supporting submodules and arbitrarily many transformer stages. Finally, a library of run-time function contracts [17] is available. Contracts are a language extension to annotate functions with arbitrary checks on input and output, with a blame system. An example of a contract is `(-> int? bool?)`, which denotes that a function must take an integer and produce a Boolean.

2) General approach: Our approach makes heavy use of the language extension capabilities of Racket, generating specialised macros from the declarations. The language extension feature enables passing the entire contents of a module to a syntax transformer. This gives the transformer absolute control over the syntax and language features for that module.

Fig. 8 illustrates the general design of our prototype. The framework provides a language for the declarations, with the keywords `define-context` and `define-controller` for specifying the system. Only these declaration-keywords are allowed as top-level terms.

When evaluated, the declarations are transformed into an application-tailored language. This language provides a keyword `implement` to provide the implementations of components. It has cases for each of the declared components. For the developer this is simple, from the framework's point of view it provides control.

3) Translation of the declarations: Here we give a general outline of the syntax that each declaration written in a Racket-DiaSpec module will be translated into, and how the `implement` macro 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 a tailored contract. Like the Java abstract method header, the type of f is derived from the interaction contract.

a) Activation conditions: These define the first argument to the function f.
`when provided x`. First argument gets type of x. For *`AssemblePic`*, the contract starts with `(-> bitmap%? ...)`.
`when required`. No argument added – the context was activated by pull.

b) Data requirements: These determine the next argument. Comparable to the Java prototype, it is a closure providing proxied access to the resource. This makes it

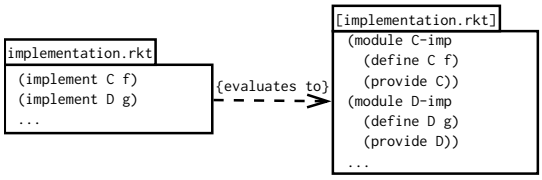

Fig. 9. Separation of components using modules. The developer's code (left), and its expanded form (right). f in C cannot access D or g, because of scoping.

```
;;; Specifications file, webcamspec.rkt
#lang s-exp "diaspec.rkt"
(define-context MakeAd String [when-required get IP])
(define-context ProcessPic Picture
  [when-provided Camera always_publish])
(define-context AssemblePic Picture
  [when-provided ProcessPicture get MakeAd maybe_publish])
(define-controller ScreenController
  [when-provided AssemblePic do Screen])
```

Fig. 10. Complete declarations of the example application, in Racket prototype.

convenient for a developer to query a resource, and allows the framework to enforce 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.

`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 return `void`.

`get nothing`. No proxy closure needed.

c) Publication requirements: These determine the last arguments to the function contract, corresponding to the return type. Publishing is treated using continuations.

`always publish`. One continuation function, *i.e.,* 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. If the developer chooses not to publish, they use the second, no-publish continuation. The contract is therefore `(-> ... (-> t? void?) (-> void?) none/c)` (the `none/c` contract accept no values: this causes a run-time error if the developer does not use one of the provided continuations.

The `implement` macro wraps 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 module. Finally, the implementation module will also be checked to contain exactly one `(implement` C `...)` for each declared C.

4) Illustration with the `AssemblePic` context: Fig. 10 is a transliteration of the declarations in Fig. 3. The developer indicates that the language to use is the one provided by `diaspec.rkt`, which provides the macros `define-context`, *etc.* This is done using the **`#lang`** `s-exp "diaspec.rkt"` tag.

```
;;; Implementation file, webcamimpl.rkt
#lang s-exp "webcamspec.rkt"
(implement AssemblePic
  (lambda (pic getAdTxt publish nopublish)
    (let* ([canvas  (make-bitmap pic ..)]
           [adtxt   (getAdTxt)])
      ; .. do magic, overlay adtxt on pic
      (publish canvas)))) ... ; more implementations
```

Fig. 11. The implementation of the `AssemblePic` context.

```
(module webcamimpl "webcamspec.rkt"
  (module AssemblePic-module racket
    (define/contract AssemblePic-impl
      (-> bitmap%? (-> string?) (-> bitmap%? void?)
          (-> void?) none/c)
      (lambda (pic getAdTxt publish nopublish)
        (let* ([canvas  (make-bitmap pic ..)]
               [adtxt   (getAdTxt)])
              ; .. do magic, overlay adtxt on pic
              (publish canvas))))
    (provide AssemblePic-impl))  ...)
```

Fig. 12. The developer's code snippet is transformed into a submodule. This is the result of evaluating Fig. 11.

When Racket encounters this annotation at the top of a file, the entire syntax tree is passed to `module-begin` in `diaspec.rkt`.

The macros we have implemented also generate a `begin-module` macro, so that the declarations module itself becomes a new Racket language. To illustrate, the implementation of *`AssemblePic`* is shown in Fig. 11. The other implementations have been omitted.

The developer simply provides a lambda term for each component. When the implementation module is loaded, the `begin-module` macro checks that all `(implement` C f`)` terms are present.

After the developer provides a conforming implementation for each component, the module can be loaded, and the complete system can be executed. To understand the run-time system, and how the requirements previously detailed are enforced, we will look at the syntax produced.

As mentioned, when the developer uses `implement`, it expands to a submodule and a contract. Fig. 12 shows the expansion of Fig. 11.

After expansion, a submodule `AssemblePic-module` is introduced. The submodule prevents its body having access to identifiers outside its lexical scope. This way 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 (see Fig. 12, line 8). There are ways a malicious developer might get around this restriction, for example by importing a module M which has shared mutable state. Contexts A and B could both import module M, and use it as a communication channel. However, since the `require` keyword in Racket is only allowed at the top-level of a module, the isolation should be sufficient. One way of circumventing this restriction could be to build an expression and import the module at run-time,

for example with *e.g.,* `(eval '(require ...))`. In fact, many nefarious things could be done this way. For this reason we have a taboo-word list. Words such as `eval` may not appear in implementations. This is not watertight, but gives a suggestion on how to mitigate such leaks.

The provided code snippet is bound to an identifier using `define/contract`, which attaches a behavioural contract to a function. It dictates that the first argument should be a picture, with `bitmap%?`. The second argument should be a closure which evaluates to a string (*i.e.,* contract `(-> string?)`). This comes 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, *c.f.* `maybe publish`.

By providing the data requirement closures, we avoid giving the developer direct access to other components. The framework proxies the call to the required components. This way, we can be sure that only legal requests are made. Alternatively, a keyword could have been introduced for this, facilitating a static verification that a resource is only polled once, for example. Finally, we see how the developer obtains and uses the provided values and publishes the computed value.

The last remaining step is collecting the implementation terms provided by the developer, allowing the framework to run the implemented system. In the generated language's `module-begin` macro, static checks are done, verifying that all the specified components are implemented exactly once. All the implementations terms like `AssemblePic-impl` are assembled in a convenience function called `run`, which the developer may call to execute the system.

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

[**Req1**] Resource usage is controlled by the framework. The developer's code is isolated with submodules and only given access to resources via checked proxy closures.

[**Req2**] By providing continuations that proxy and check publications, we can be sure that the developer cannot influence the control flow. Combined with submodule scoping this ensures deep separation of components.

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

[**Req4**] The structure of the implementation is verified statically, but the types 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 [15], the checks would be static, like in Java. Using Typed Racket would be trivial – it amounts to changing the language to `typed/racket` instead of `racket` (in Fig. 12, on line 2), and translating the contract-syntax into the very similar Typed Racket syntax (line 4).

We might later consider adding a safe way for developers to specify which libraries they would like to use, since currently only Racket base is provided. It might be unfortunate that we disallow importing of modules if a developer desires a particular library. It would be easy to provide our own `require`-like keyword, although this might allow aforementioned illicit communication (breaking **Req2**).

It is not essential to use the language extension feature of Racket, 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 require that only `implement` terms appear at the module level, or that the declarations only consist of uses of `define-context`. This also enables the taboo-word check.

We could also have refrained from using macros, simply manipulating plain data structures – a non-embedded DSL approach. Apart from the inconvenience of not having specialised syntax, this would have the disadvantage that checks such as coherence of the implementation, existence of the required implementations, *etc.* would all be dynamic. Because we value early warnings for the developer, we prefer our solution where the developer's mistakes cause the syntax check to fail.

IV. Principles

Here we try to synthesise principles for how to implement the general requirements relevant to open platforms outlined in Sec. II-A, in arbitrary programming languages. We try to generalise the case study to framework design and how requirements translate to programming features in general, to help understand and build such frameworks.

A. Treating declarations statically or dynamically influences restriction and guidance

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. This need can be fulfilled by declaration-driven frameworks. For each requirement and for each resource the framework developer may choose to handle the restrictions either statically or dynamically, depending on the sensitivity of the resource. Note that it is not necessary to choose one approach globally.

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 will be used. Consider the case study, where a developer cannot access resources illegally, without triggering compile-time errors. If a dynamic approach is chosen, an advantage is a high degree of accuracy regarding which resources are used, and when. However, this means receiving late warnings about incorrect code.

For example, in iOS the dynamically-handled resource access controls are the most accurate: the user can choose to allow or disallow on a per-resource basis, if and when access is requested. This still does 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, Android's permissions are also dynamically checked, but even less favourably. If a given resource remains unused for a particular session it still has to be allowed by the user at install time. This is a vulnerability, *e.g.,* advertisement libraries abuse their embedding into over-privileged applications [9], [10]. In DiaSuite, the permissions are also fixed once for all sessions, but are granted per-component. This is used as an advantage: the developer can be helped by specialising the API per application, and giving the developer warnings about misuse of resources at compile time. Crucially, the user also has more insight into data reachability. One might consider making a veto for individual resources possible, too.

A recent version[5] of Android introduced the possibility of granting an application a subset of its requested permissions, *e.g.,* GPS accessible but contact list forbidden. This evolution highlights that the old permission model was not meeting users' needs. This suggests that the original design choices were ad-hoc, with implicit principles – now that there is a large user base, feedback is appearing on how this situation should be improved. We hope that our work can clarify the possible design space for such systems. Also, it might promote a better understanding of the choices of static *vs.* dynamic enforcing of particular rules, including advantages and trade-offs. Unfortunately, from the point of view of parties earning ad revenue, having access to private user data is in fact a selling point. This implies that platform owners have a certain incentive to keep platforms "leaky". If not, part of their valuable market share might be lost to even less scrupulous platform providers.

B. 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 encode 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 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.

In fact, all we need is a programming environment with at least one stage before run-time, enabling processing of the static semantics of the declarations. Consider the Racket example, where we make no use of static features or a type system, but implement everything using syntax transformers, which can simply be seen as extra compilation phases. A type system can be considered a limited-expressiveness programming stage, which is how the properties are verified in the Java prototype.

If we have (or can implement, whether by a declaration compiler or syntax transformers) 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 macro expansion for the Racket implementation for achieving this kind of checking. Therefore, widely varying techniques can be used to implement stages. A strong and/or static type system is thus not required, even if static enforcing is desired.

V. Discussion

In this section we discuss the lessons that we have learned from the implementation of the prototypes in Java and Racket.

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 tailored frameworks can deliver great advantages.

We emphasise that from the developers' and users' point of view, both the Java and Racket prototypes are equivalent in terms of features and guarantees. This is a convincing argument for the fact that a strong type system is no prerequisite for a property-ensuring framework. Most of the verification is static, even in Racket, because of our use of syntax transformers. This enables raising early errors and warnings at syntax-checking time. However, Racket is a very extensible and expressive language, which might give an optimistic impression of what can be achieved in other, less expressive, host languages.

A. A strong case for declaration-driven tailored frameworks

Both in object-oriented and functional settings, we see that declaration-driven frameworks potentially turn declarations from an advisory document full of promises into verifiable properties. This can provide the user of the application with valuable information on the behaviour of an application. Also, we see that from the developer's point of view, implementing an application using a tailored framework is less laborious than using a general-purpose framework. In our example, only the components need to be implemented. All communication, deployment, *etc.* is taken care of by the framework. This is possible because the framework has detailed information about the intended structure and behaviour of the app.

Therefore, we believe that this new generation of frameworks can provide fundamental advantages. Even if general-purpose frameworks do provide a notion of restrictions, the available declarations (*e.g.,* the Manifest file in Android) are not used to ease development. This seems like a missed opportunity.

[5]As of version 4.3, Android contains a hidden screen to (dis)allow access per-application and per-resource [18].

We also find that the implementations in both Java and Racket are natural. Extending classes is the traditional way of using an object-oriented framework, as is the use of domain-specific languages to solve problems in a Lisp-like language. Even while using traditional approaches, it seems feasible to go beyond the status quo and enforce properties, give the user insight into the usage of their resources, and assist the developer. We would argue, though, that the Racket prototype, from the developers' point of view, probably is even more natural. There, the tailored language makes it very simple to provide an implementation. In Java, on the other hand, there is the relatively unwieldy proxy classes, plus the deployment boilerplate code (Fig. 7), which might be a stumbling block to developers. We conjecture that this is because functional languages with higher-order functions allow a more natural encoding of inversion of control concepts than is possible in Java.[6]

B. 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.

1) DiaSuite, Yesod: This work was inspired by DiaSuite [6], [11]. Therefore, it most closely resembles the prototype. However, while the articles related to DiaSuite explain the theory of interaction contracts, and the idea of a generated tailored framework which guides and supports the developer, the reflection on design space, as well as what the requirements are for implementing such a system, have not been addressed. Furthermore, the discussion about DiaSuite's design 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 [5] 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. 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.

2) TouchDevelop: 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.* [20] provide a domain-specific programming language based on TouchDevelop[7] [21] to facilitate static analysis, per-resource permissions, and showing a user what the potential flow of information is (*e.g.,* camera → 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.*). It would be relatively simple to extract such "arrows" from our app specifications.

3) TaintDroid: The authors of TaintDroid [22] propose another novel approach: real-time taint analysis run in parallel on a remote server. This approach is probably 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.

4) Non-invasive static analysis: Much other work exists, including specialised work on Android [23]–[27], looking into static analysis of existing code without changing 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. In comparison, we require modification of the platform, but believe this is justified by gains in terms of privacy for the user.

VI. 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, this could be modelled 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 the possibilities and advantages for functional programmers provided by this technique, as well as establishing good practices for developing frameworks for open platforms.

[6]Java 8 includes lambda functions [19]. They could provide a more elegant way of modelling the proxied polling access to resources, as done in Racket.

[7]TouchDevelop [21] is an application creation environment allowing developers to write scripts for mobile devices and publish them in an app store for users to install. 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 anonymised values.

REFERENCES

[1] R. Prieto-Diaz and P. Freeman, "Classifying software for reusability," *Software, IEEE*, vol. 4, no. 1, pp. 6–16, 1987.

[2] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[3] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Commun. ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997. [Online]. Available: http://doi.acm.org/10.1145/262793.262798

[4] R. Rogers, J. Lombardo, Z. Mednieks, and B. Meike, *Android Application Development: Programming with the Google SDK*. Beijing: O'Reilly, 2009.

[5] M. Snoyman, *Developing Web Applications with Haskell and Yesod – Safety-Driven Web Development*. O'Reilly, 2012.

[6] D. Cassou, J. Bruneau, C. Consel, and E. Balland, "Toward a tool-based development methodology for pervasive computing applications," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1445–1463, 2012.

[7] J. L. Dave Mark, *Beginning iPhone Development: Exploring the iPhone SDK*. Apress, 2009.

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

[9] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the Android ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 31–40.

[10] A. Bartel, J. Klein, Y. Le Traon, and M. 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*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 274–277. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351722

[11] D. Cassou, E. Balland, C. Consel, and J. 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*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 431–440. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985852

[12] M. Flatt and PLT, "Reference: Racket," PLT Inc., Tech. Rep. PLT-TR-2010-1, 2010, http://racket-lang.org/tr1/.

[13] N. Cameron and J. Noble, "Encoding ownership types in Java," in *Objects, Models, Components, Patterns*. Springer, 2010, pp. 271–290.

[14] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, 1986, pp. 151–161.

[15] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *ACM SIGPLAN Notices*, vol. 46. ACM, 2011, pp. 132–141.

[16] M. Flatt, "Submodules in Racket: You want it when, again?" in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, ser. GPCE '13. New York, NY, USA: ACM, 2013, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2517208.2517211

[17] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen, "Correct blame for contracts: no more scapegoating," in *ACM SIGPLAN Notices*, vol. 46. ACM, 2011, pp. 215–226.

[18] "Hidden Android feature allows users to fine tune app permissions," Online, http://www.zdnet.com/hidden-android-feature-allows-users-to-fine-tune-app-permissions-7000018944/, 2013, accessed: 2014-03-23.

[19] Oracle, "Learning the Java Language: Lambda Expressions," Mar. 2014. [Online]. Available: http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[20] X. Xiao, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal, "User-aware privacy control via extended static-information-flow analysis," in *ASE*, M. Goedicke, T. Menzies, and M. Saeki, Eds. ACM, 2012, pp. 80–89.

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

[22] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.

[23] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang, "A static assurance analysis of Android applications," *Virginia Polytechnic Institute and State University, Tech. Rep*, 2013.

[24] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in Android applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1457–1462.

[25] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for Android applications," EC SPRIDE, Tech. Rep. TUD-CS-2013-0113, May 2013.

[26] C. Gibler, J. Crussel, J. Erickson, and H. Chen, "AndroidLeaks: Detecting privacy leaks in Android applications," Tech. rep., UC Davis, Tech. Rep., 2011.

[27] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2382756.2382798