SeMA: Extending and Analyzing Storyboards to Develop Secure Android Apps

Joydeep Mitra
Kansas State University, USA
joydeep@ksu.edu

Venkatesh-Prasad Ranganath
rvprasad.free@gmail.com

Torben Amtoft
Kansas State University, USA
tamtoft@ksu.edu

Michael Higgins
Kansas State University, USA
mikehiggins@ksu.edu

July 20, 2020

Abstract

Mobile apps provide various critical services, such as banking, communication, and healthcare. To this end, they have access to our personal information and have the ability to perform actions on our behalf. Hence, securing mobile apps is crucial to ensuring the privacy and safety of its users.

Recent research efforts have focused on developing solutions to secure mobile ecosystems (i.e., app platforms, apps, and app stores), specifically in the context of detecting vulnerabilities in Android apps. Despite this attention, known vulnerabilities are often found in mobile apps, which can be exploited by malicious apps to harm the user. Further, fixing vulnerabilities after developing an app has downsides in terms of time, resources, user inconvenience, and information loss.

In an attempt to address this concern, we have developed SeMA, a mobile app development methodology that builds on existing mobile app design artifacts such as storyboards. With SeMA, security is a first-class citizen in an app's design – app designers and developers can collaborate to specify and reason about the security properties of an app at an abstract level without being distracted by implementation level details. Our realization of SeMA using Android Studio tooling demonstrates the methodology is complementary to existing design and development practices. An evaluation of the effectiveness of SeMA shows the methodology can detect and help prevent 49 vulnerabilities known to occur in Android apps. Further, a usability study of the methodology involving ten real-world developers shows the methodology is likely to reduce the development time and help developers uncover and prevent known vulnerabilities while designing apps.

1 Introduction

1.1 Motivation

Android apps help us with critical tasks such as banking and communication. Consequently, they need access to our personal information. Hence, developers try to ensure that their apps do not cause harm to the user. Despite such efforts, apps exhibit vulnerabilities that can be exploited by malicious apps either installed in the device or executing remotely. In 2018, an industrial study of 17 fully functional Android apps discovered that all the apps had vulnerabilities [46]. 43% of the vulnerabilities were classified as high risk, and 60% of them were on the client-side. In 2019, a vulnerability in Google's camera app allowed a malicious app without required permissions to gain full control of the camera app and access the photos and videos stored by the camera app [6]. A recent study showed that approximately 11% of 2,000 Android apps collected from a couple of app stores, including Google Play, are vulnerable to Man-In-The-Middle (MITM) and phishing attacks [50]. Further, a vulnerability in numerous real-world apps was recently discovered (e.g., the GMail app on Android) that allowed a malicious app to exploit the vulnerability and carry out phishing and denial-of-service attacks on those apps [1].

In the last decade, researchers have developed a plethora of tools and techniques to help detect vulnerabilities in Android apps [45, 41, 28]. Even so, apps with vulnerabilities find their way to app stores because app developers do not use these tools, or the tools are ineffective. The latter reason is supported by Ranganath and Mitra [38]. Additionally, Pauck et al. [36] assessed six prominent taint analysis tools aimed at discovering vulnerabilities in Android and found tools to be ineffective in detecting vulnerabilities in real-world apps. In a similar vein, Luo et al. [31] conducted a qualitative analysis of Android static taint analysis tools and found that these tools not only miss certain taint flows since they do not consider an app's context, but their default configurations lead to many false positives.

Moreover, existing approaches to secure Android apps are curative; that is, they are aimed at detecting vulnerabilities after they are introduced during implementation. Identifying and fixing vulnerabilities after they occur in implementation increases the cost of development. Consequently, there has been a growing call to integrate security into every phase of the software development life-cycle. In a 2014 study, Green and Smith [14] argued that to build secure software, developers need support in various areas ranging from safer programming languages to better security testing tools. Therefore, given the current landscape of Android app security and to support the call for secure software development, we are exploring a preventive approach that can help prevent the occurrence of vulnerabilities in app implementations (as opposed to curing vulnerabilities in app implementations).

1.2 Contributions

Motivated by the above observations, we make the following contributions in this article:

1. We propose SeMA, a development methodology based on an existing mobile app design technique called *storyboarding*. We describe the methodology along with the techniques used to realize it.

2. We demonstrate the effectiveness of SeMA – *can SeMA detect and help prevent vulnerabilities?* – by applying it to 49 known Android app vulnerabilities that are captured in the Ghera benchmark suite and are likely to occur in real-world apps. Our findings show SeMA is highly effective in detecting these vulnerabilities.

3. We demonstrate the usability of SeMA by having developers build Android apps with features commonly found in real-world apps. Our findings suggest, with basic interventions, SeMA is highly effective in helping developers detect and prevent the vulnerabilities targeted by SeMA without increasing the design and development times.

In the rest of the paper, all our remarks about SeMA's ability to prevent vulnerabilities are limited to the 49 vulnerabilities targeted by SeMA.

2 Background

SeMA borrows heavily from Model Driven development to iteratively build an app from its storyboard, an existing design artifact.

Model Driven Development (MDD) In MDD, software is developed by iteratively refining abstract and formal models [4]. A model is meant to capture the application's behavior and is expressed in a domain-specific language (DSL). Apart from models, the domain-specific platform is a crucial entity in MDD. The domain-specific platform provides frameworks and APIs to enable easy refinement of a model into a platform-specific implementation. Since every aspect of an application can seldom be specified in the DSL, the resulting implementation is often extended with additional code; mostly, the business logic of the application.

Traditionally, the software engineering community has been hesitant to adopt MDD due to social, cultural, and technical reasons [43]. However, with the relatively recent development of sophisticated tools that enable MDD, software engineering teams are more open to embracing it. For example, Amazon uses TLA+ to develop web services [27, 51]. Tools like Alloy [24], UML/OCL [40], and UMLSec [26] help create and analyze models of software behavior, which form the basis of further development.
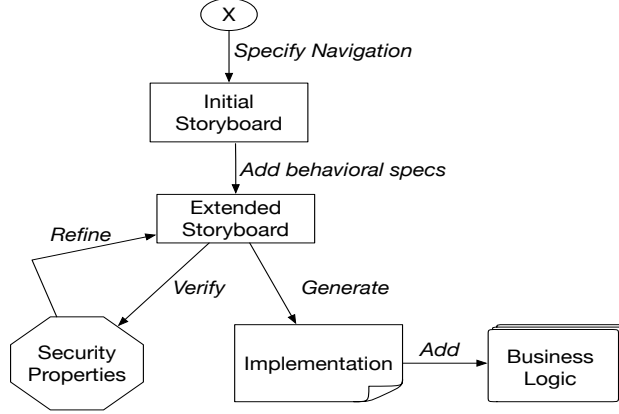
Figure 1: Development Process in SeMA

Storyboarding Android app development teams use storyboards to design an app's navigation [37, 29]. A storyboard is a sequence of screens and transitions between the screens. It serves as a model for the user's observed behavior in terms of screens and transitions. Numerous tools such as Xcode [2], Sketch [44], and Android's JetPack [13] help express a storyboard digitally. The storyboarding process is *participatory* in nature because it allows designers to get feedback from potential users about the features captured in the storyboard and from developers about the feasibility of implementing those features. However, traditional storyboards cannot capture an app's behavior beyond UI and navigational possibilities. This limitation hinders the possibility of collaboration between designers and developers while specifying the storyboard.

MDD with Storyboarding Existing MDD approaches to mobile app development do not use storyboards and do not consider security requirements. Extending storyboards with capabilities to capture an app's behavior to enable MDD has numerous benefits. First, since storyboards are becoming an integral part of the mobile app development process [47], MDD based on storyboarding is not likely to disrupt the current mobile app development process. Second, an extended storyboard can serve as a common substrate for collaboration between designers and developers to specify an app's behavior along with its UI and navigational features. Third, an extended storyboard can serve as a model of the app to formally analyze its behavior. Further, the model's abstractness reduces the cost of such analysis as it eliminates the need for and the associated cost of extracting high-level (abstract) behaviors captured in code. Finally, the storyboard can serve as a reference for an app's behavior when auditing the app's implementation.

3   The Methodology

SeMA enables the reasoning and verification of security properties of an app's storyboard via iterative refinement. A schematic of the process of developing an app with SeMA is shown in Figure 1. The process begins with a developer sketching the initial storyboard of an app, as shown in Figure 2. The developer then extends the storyboard with the app's behavior and checks if the behaviors satisfy various pre-defined security properties. The developer may repeat the previous steps to revise and refine the behaviors. Once the storyboard has captured the behavior as intended by the developer while satisfying pre-defined security properties, the developer generates an implementation from the storyboard with a push of a button. As the final step, the developer adds business logic to the implementation via hooks provided in the generated code.

3.1   Extended Storyboard

A *traditional storyboard* used in the design of mobile apps is composed of screens and transitions between screens. A screen is a collection of named widgets that allow the user to interact with the app, e.g., clicking
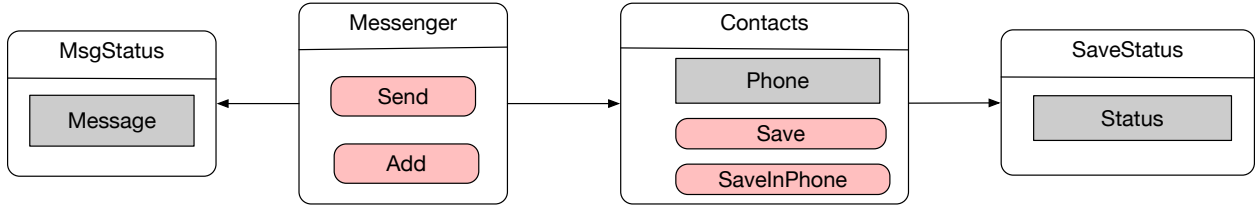
Figure 2: Diagrammatic representation of an initial storyboard of an app in SeMA, which is also a traditional storyboard.
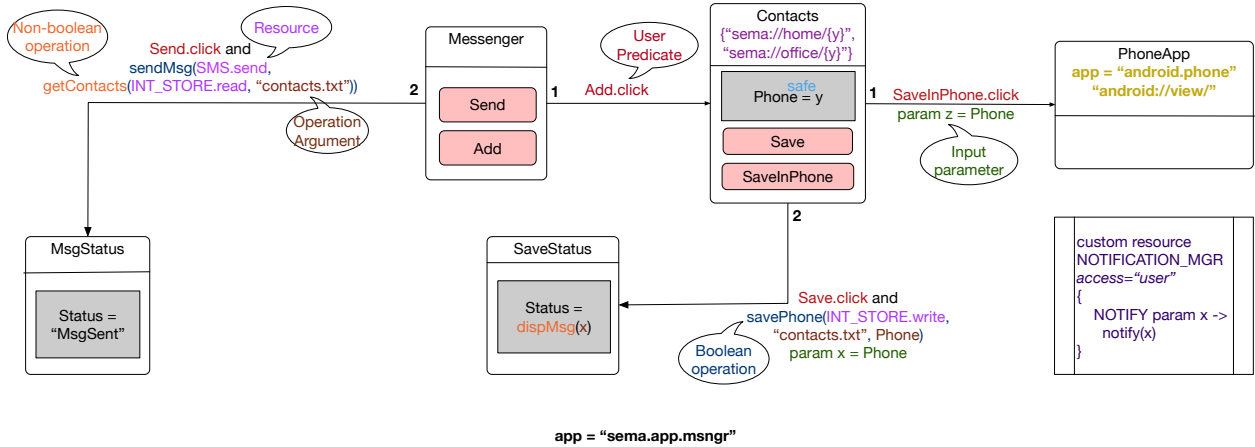

Figure 3: Diagrammatic representation of an extended storyboard. The (information) bubbles are not part of the storyboard.

a button. A transition (edge) between two screens depicts a navigation path from the source screen to the destination screen. The basic structure of a storyboard defines the navigational paths in an app.

For example, Figure 2 shows the traditional storyboard of an app with four screens: `Messenger`, `Contacts`, `MsgStatus`, and `SaveStatus`. Starting from the *Messenger* screen, a user can either add contact numbers to the app via `Contacts` screen or send a message to all saved contact numbers.

A traditional storyboard does not support the specification of app behavior. Hence, we propose the following extensions to traditional storyboards to enable the specification of app behaviors in storyboards (i.e., enrich a traditional storyboard, as in Figure 2) into an extended storyboard, as in Figure 3. *These figures are diagrammatic representations of storyboards (known as navigation graphs) in Android Studio.*

App Identity Every storyboard has an *app* attribute, which is a unique string constant used to identify the app described by the storyboard.

Extensions to Screens Screens are extended with a mandatory *name*, optional *input parameters* (in green in Figure 3), and optional *URIs* (in purple in Figure 3).

The *Input parameters of a screen* are similar to parameters of a function in mainstream programming. Input parameters bind to the values (arguments) provided when the screen is activated by either another screen in the app via an incoming transition or an app via the screen's *URI*.

The *URIs* associated with a screen can be used to access the screen from outside the app. A URI can have input parameters; similar to query parameters in web URLs. URI input parameters serve as input parameters of the screen. All URIs associated with a screen must have the same set of input parameters. For example, both URIs associated with the `Contacts` screen in Figure 3, have the input parameter y. External apps accessing a screen via its URI must provide the arguments corresponding to the URI's input parameters. Every URI without its parameters must be unique in an app.

Proxy Screens of External Apps   Apps often interact with external apps. To capture this interaction, *proxy screen* representing a screen of an external app can be included in an extended storyboard of an app; see `PhoneApp` in Figure 3. Such proxy screens have a mandatory *name*, a mandatory *URI*, and an optional *app* attributes. If *app* is specified in a proxy screen, then the proxy screen denotes the screen identified by the *URI* in the app named *app*. If *app* is not specified, then the proxy screen denotes any screen identified by the *URI* in an app installed on the device and determined by Android.

Extensions to Widgets   Widgets are extended with a mandatory *value* that can be assigned by the developer (e.g., in labels), entered by the user (e.g., in fields), provided by an input parameter of the containing screen (e.g., when the screen is activated by a transition), or returned by an operation. Based on the displayed content, widgets can be of different types, e.g., a label and a text widget display plain text while a web widget displays web content. Further, depending on the widget's type, a widget can be configured with a pre-defined set of rules that regulate the data displayed in a widget, e.g., a *WebView* widget can be configured with a whitelist of trusted URL patterns (via *trusted-patterns* attribute) to display content only from URLs in the whitelist.

Resources   Android provides apps with resources with different capabilities, e.g., storage, networking. Hence, to complement this aspect, storyboards are extended with a pre-defined set of resources with specific capabilities that can be used by the apps being designed. [1]   Android apps can offer UI-less services to other apps, e.g., broadcast receivers, content providers. Such services are denoted by custom resources in storyboards. A *custom resource* offers capabilities that can be used by apps installed on the device. Each custom resource has a mandatory identifier that is unique to the app. Each capability of a resource has a mandatory identifier that is unique to the resource. Also, each capability that has security implications can be marked as *privileged*. Access to a resource and its capabilities can be controlled via the *access* attribute of the resource. This attribute can take on one of the following three values: *all* indicates any app can access the resource/capability, *user* indicates user permission is required to access the resource/capability, and *own* indicates only the resource defining app x or an app that shares the digital signature of app x can access the resource/capability. For example, in Figure 3, NOTIFICATION_MGR is a custom resource that offers NOTIFY capability with a `notify` operation. Based on its access attribute, a client will need to seek the user's permission to use its NOTIFY capability. These values of the *access* attribute are based on permission levels supported by Android.

Operations   In an extended storyboard, an *operation* indicates a task to be performed, e.g., read from a file, get contents from a web server. An operation has a *name*, returns a value, may have *input parameters*, and may use a *capability* (provided by a resource). An operation is used by mentioning its name along with arguments and any required capabilities. For example, in Figure 3, operation `savePhone` is used to save data in the device's internal storage by using the *write* capability of the internal storage device exposed as the resource *INT_STORE*. A use of an operation introduces (declares) it in the storyboard. An operation is defined in the generated implementation (described in Section 3.4). Use of operations *must* be consistent; that is, a non-boolean operation cannot be used in a boolean value position.

Extension to Transitions   Transitions between screens can be adorned with constraints that, when satisfied, enable/trigger a transition. A constraint is a conjunction of a *user action* (e.g., click of a button) and a *boolean expression*, comprising a combination of boolean operations. A constraint is satisfied when the user action is performed (e.g., *save* button is clicked) and the boolean expression evaluates to true. For example, in Figure 3, the transition from `Contacts` screen to `SaveStatus` screen is taken only when the user clicks the `Save` button and the `savePhone` operation evaluates to *true*; that is, the value of `Phone` is successfully saved in "contacts.txt", a file on the internal storage of the device.

  As part of a transition, arguments are provided to the input parameters of the destination screen (in green in Figure 3). An argument can be a literal specified in the storyboard, a value available in the source screen of the transition (e.g., value of a contained widget, input parameter to the screen), or a value returned by an operation. Further, every transition to a screen must provide values (arguments) for every input parameter

---

[1]The current realization of SeMA supports a subset of pre-defined resources offered by Android.

of that screen. For example, if there are two transitions *t1* and *t2* to screen *s* with input parameters x and y, then arguments for both x and y must be provided along both *t1* and *t2*.

Multiple outgoing transitions from a screen may be simultaneously enabled when their constraints are not mutually exclusive. Hence, to handle such situations, all outgoing transitions from a screen must be totally ordered; see the number labels on outgoing transitions from `Contacts` screen in Figure 3. The implementation derived from the storyboard will evaluate the constraints according to the specified order of transitions and take the first enabled transition.

The formal description of the syntax and semantics of the extended storyboard is provided in Appendix A and B, respectively.

3.2 Security Properties

An Android app often interacts with other apps on the device, the underlying platform, and remote servers. Such interaction involves sharing of information, responding to events, and performing tasks based on user actions. Many of these interactions have security implications that should be considered during app development, e.g., can the user's personal information be stored safely on external storage? (information leak and data injection), who should have access to the content provided by the app? (permissions and privilege escalation), how should an app contact the server? (encryption).

While security implications are relevant, our current effort focuses on implications related to confidentiality and integrity of data.

Confidentiality *An app violates confidentiality if it releases data to an untrusted sink.* Hence, an app (and, consequently, its storyboard) that violates confidentiality is deemed as insecure. We explain the concepts of untrusted sinks in a storyboard in Section 3.3.

Integrity *An app violates integrity if it uses a value from an untrusted source.* Hence, an app (and, consequently, its storyboard) that violates integrity is deemed as insecure. We explain how a source is identified as untrusted in Section 3.3.

3.3 Storyboard Analysis

There are multiple ways to check if extended storyboards satisfy various properties. In our current realization of SeMA, we use *information flow analysis* and *rule checking* to check and help ensure extended storyboards satisfy security properties concerning confidentiality and integrity.

3.3.1 Information Flow Analysis (IF)

This analysis tracks the flow of *information* in the form of values from sources to sinks in a storyboard.

A *source* is either a widget in a screen, an input parameter of a screen, or (the return value of) an operation. The set of sources in a storyboard are partitioned trusted and untrusted sources based on the guarantee of data integrity. Specifically, a source is *untrusted* if it is an input parameter of a screen's URI or it is an operation that reads data from an HTTP server, an open socket, device's external storage, or device's clipboard; or uses the capability of a resource provided by another app. All other sources are deemed as trusted.

A *sink* is either a widget in a screen or an argument to a screen or an operation. The set of sinks in a storyboard are partitioned trusted and untrusted sinks based on the guarantee of data confidentiality. Specifically, a sink is deemed as *untrusted* if it is an argument to an external screen identified without the *app* attribute or to an operation that writes data to an HTTP server, an open socket, device's external storage, or device's clipboard; or uses the capability of a resource provided by another app. All other sinks are deemed as trusted.

The analysis relies on other actors (e.g., developer) and systems (e.g., user authentication and key management by Android runtime platform and app validation and authentication by app stores) to ensure aspects such as authenticity and trust of sources and sinks.

To reason about the flow of information between sources and sinks, we define a binary reflexive relation named *influences* between sources and sinks as follows: *influences(x,y)* (i.e., source x influences sink y) if

1. x is assigned to an input parameter y of screen s on an incoming transition to s,

2. x is an argument to operation y, or

3. value of operation x or input parameter x of screen s is assigned to widget y in screen s.

Here are few instances of this relation in Figure 3. *influences(y, Phone)* because input parameter y of `Contacts` screen is assigned to `Phone` widget. *influences(x,dispMsg)* because x is provided as an argument to operation `dispMsg`. *influences(dispMsg, Status)* because the value of operation `dispMsg` is assigned to `Status` widget in `SaveStatus` screen.

Further, all data flows inside the app are guaranteed to preserve the confidentiality and integrity of used/processed data. The operations provided as part of the capabilities of custom resources provided by the app are assumed to preserve the confidentiality and integrity of used/processed data.

With the above (direct) influence relation, guarantees, and assumptions, we use the transitive closure of *influences* relation to detect violation of security properties. We detect *potential violation of integrity* by identifying transitive (indirect) influences $(x, y) \in influences^*$ in which x is an *untrusted* source. Likewise, we detect *potential violation of confidentiality* by identifying transitive (indirect) influences $(x, y) \in influences^*$ in which y is an *untrusted* sink. All such identified indirect influences are reported to the developer and must be eliminated. Such an indirect influence can be eliminated by either replacing untrusted sources/sinks with trusted sources/sinks or indicating the indirect influence as safe by marking one or more of the direct influences that contribute to the indirect influence as *safe*.[23]

For example, in Figure 3, the y parameter of `Contacts` screen is untrusted as it is provided by an external app. So, the analysis will flag $influences^*(y, Phone)$ as violating integrity due to the assignment of y to `Phone` widget in `Contacts` screen. A developer can fix this violation either by removing the assignment of y to `Phone` or marking the assignment as *safe* (as done in Figure 3).

Correctness of the analysis The purpose of the analysis is to identify violations of confidentiality or integrity. This is done in two steps: 1) calculate the potential flows in the storyboard using the transitive closure of *influences* relation and 2) check if a flow involves an untrusted source or sink and is not marked as *safe*. Since the second step is based on pre-defined classification of sources and sinks and developer-provided *safe* annotations, the correctness of the analysis hinges on the first step.

If the *influences* relation correctly captures the direct flow between sources and sinks in a storyboard, then the transitive closure *influences** will capture all possible flows between the sources and sinks, including all flows violating confidentiality or integrity. Hence, *the analysis is complete* in identifying every violation of confidentiality or integrity.

The *influences** relation does not consider the effect of constraints on flows between sources and sinks; that is, all constraints on transitions are assumed to be true. Consequently, the analysis may identify a flow between a source and a sink when there is no flow between the source and the sink (at runtime). For example, suppose a screen s has two incoming transitions i_1 and i_2 and two outgoing transitions o_1 and o_2 along with transition constraints that dictate transition o_x must be taken if and only if s was reached via transition i_x. Further, suppose an input parameter m of s is assigned a value along i_1 and i_2 and used as an argument along o_1 and o_2. In this case, the value assigned to m along i_1 (i_2) will not flow out of m along o_2 (o_1). However, the analysis will incorrectly identify the value assigned to m along i_1 (i_2) may flow out of m along o_2 (o_1). Since the analysis may report invalid violations, *the analysis is unsound*. However, a developer can override such violations by marking relevant direct flows as *safe*.

A formal description of the information flow analysis along with the algorithm and its correctness proof is provided in Appendix C.

---

[2]This marking is similar to data declassification in traditional information flow analysis.

[3]When an indirect influence is enabled via multiple sequences of direct influences between a source and a sink (similar to multiple paths between nodes in a graph), at least one direct influence in each sequence should be marked as safe to indicate the indirect influence is safe.

3.3.2 Rule Checking (RC)

Prior research has developed guidelines and best practices for secure Android app development [35, 23]. Based on these standards, we have developed rules that can be enforced at design time (in addition to information flow analysis) to prevent the violation of properties related to confidentiality and integrity.

Following is the list of rules supported by the current realization of SeMA, along with the reasons for the rules.

1. *Capabilities offered by custom resources must be protected by access control.* If any external client can access a custom resource (i.e., its *access* attribute is set to *all*) and the resource offers privileged capabilities, then malicious clients can gain access to privileged capabilities without the user's consent. Further, *Android's policy of least privilege* stipulates that apps should have minimal privileges and acquire the privileges required to use protected services.

2. *WebView widgets must be configured with a whitelist of URL patterns.* A WebView widget in an app works like a browser – it accepts a URL and loads its contents – but it does not have many of the security features of full-blown browsers. Also, a WebView widget has the same privileges as the containing app, has access to the app's resources, can be configured to execute JavaScript code. Hence, loading content from untrusted sources into WebView widgets facilitates exploitation by malicious content.

3. *Operations configured to use HTTPS remote servers must use certificate pinning.* HTTPS remote servers are signed with digital certificates issued by certificate authorities (CAs). Android defines a list of trusted CAs and verifies that the certificate of an HTTPS remote server is signed with a signature from a trusted CA. However, if a trusted CA is compromised, then it can be used to issue certificates for malicious servers. Hence, to protect against such situations, certificates of trusted servers are pinned (stored) in apps, and only servers with these certificates are recognized as legit servers by the apps.

4. *Operations configured to use SSL sockets must use certificate pinning.* The reasons from the case of certificate pinning for HTTPS applies here as well.

5. *Cipher operations must use keys stored in secure key stores (containers).* The results of cipher operations can be influenced by tampering the cryptographic keys used in cipher operations. Further, since cryptographic keys are often used across multiple executions of an app, they need to be stored in secondary storage that is often accessible by all apps on a device. Hence, to protect against unwanted influences via key tampering, cipher keys should be stored in secure key stores (containers).

Realization of Rule Checking   Violations of rule 1 are detected by checking if a custom resource offers a privileged capability and has its *access* attribute set to *all*.

The *trust-patterns* attribute of *WebView* widget is used to specify the whitelist of trusted URL patterns. Violations of rule 2 is detected by checking if *trust-patterns* attribute is specified for every *WebView* widget.

Violations of rule 5 are detected by checking if the key argument provided to a cipher operation is the value returned by a pre-defined operation to keys from a secure container.

Violations of rules 1, 2, and 5 are flagged as errors and must be addressed before moving to the code generation phase.

Certificate pinning is enabled by default in every storyboard in the methodology. However, since techniques other than certificate pinning can be used to secure connections to servers, a developer can disable certificate pinning by setting *disableCertPin* attribute in a network-related operation. Such cases are detected as violations of rules 3 and 4. They are flagged as warnings but do not inhibit the developer from moving to the code generation phase.

3.4 Code Generation

Once the developer has verified that the specified storyboard does not violate properties related to confidentiality and integrity, she can generate code from the storyboard. Figure 4 shows a fragment of generated code for the running example.

```
18   public class ContactsFrag extends Fragment {    Contacts Screen
19     private EditText phoneNumObj;
20     private ContactsFragbusLogic objBusLogic = new ContactsFragbusLogic();
21     @Override
22     public final View onCreateView(LayoutInflater inflater, ViewGroup container,
23         Bundle savedInstanceState) {
24       return inflater.inflate(R.layout.contacts_frag, container, false);
25     }
26     private boolean savePhone(String filePath, String param1, String phoneNum) {
27       try(FileOutputStream outputStream = getContext().openFileOutput(filePath, Context.MODE_PRIVATE)) {
28         objBusLogic.busLogicsavePhone(outputStream,param1,phoneNum);
29       }
30       catch(IOException e) {
31         e.printStackTrace();        Boolean operation that uses INT_STORE resource
32         return false;                to save phone number in a file in internal storage.
33       }
34       return true;
35     }
36     @Override
37     public final void onViewCreated(View view, Bundle savedInstanceState) {
38       super.onViewCreated(view, savedInstanceState);
39       phoneNumObj = (EditText) view.findViewById(R.id.phoneNum);      Constraint based on user's
40       view.findViewById(R.id.save).setOnClickListener(new View.OnClickListener() {   action on the button save.
41         @Override
42         public void onClick(View v) {                           Transition to MsgStatus screen
43           if (savePhone("contacts.txt",null,phoneNumObj.getText().toString())) {   with "x" as argument.
44             Bundle destArgs = new Bundle();
45             destArgs.putString("x",phoneNumObj.getText().toString());
46             Navigation.findNavController(getView()).navigate(R.id.goTostatus, destArgs, null);
47           }
48         }
```

Figure 4: Code generated for the Contacts screen in the storyboard depicted in Figure 3

The current realization of SeMA hinges on various choices in mapping and translating storyboard-level entities and concepts into code-level entities and concepts. These choices are encoded in the following mapping and translation rules used during code generation.

1. A Screen is translated to a `Fragment`. For each input parameter of the screen, a function to obtain the value of the parameter is generated. If an input parameter is not available at runtime, then the corresponding function raises a runtime exception.

2. A widget is translated to the corresponding widget type in Android, e.g., a widget displaying text is translated to `TextView`. The value of the widget is the corresponding value specified in the storyboard. For example, if the value is provided by a screen's input parameter x, then the return value from the getter function of x is set as the widget's value, e.g., `TextView.setText(getX())`. The value in a widget is obtained via the corresponding getter function, e.g., `TextView.getText()`.

3. The constraint associated with a transition from a source screen to a destination screen is a conjunction of a user action and boolean operations. The user action part of the constraint is translated to a listener/handler function in the source screen that is triggered by the corresponding user action, e.g., button click. If the constraint has a boolean expression, then a conditional statement is generated with the boolean expression as the condition in the body of the listener function. The *then* block of the conditional statement has the statements required to trigger the destination screen. If the constraint has no boolean operations, then the body of the listener function has statements required to trigger the destination screen. If the constraint has no user action, then the checks corresponding to the boolean operations are performed when the source fragment is loaded.

   We use Android's navigation APIs to trigger a destination screen. If the destination screen is a proxy screen, then intents are used to trigger the destination screen determined by the *URI* and *app* attribute. Arguments to destination screens are provided as key/value pairs bundled via the `Bundle` API.

   When a screen has multiple outgoing transitions, the statements corresponding to the transitions are chained in the specified order of the transitions in the storyboard.

4. An operation is translated to a function with appropriate input parameters and return value. Each reference to the operation in a storyboard is translated to call the corresponding function.

   The type of the input parameters depends on the type of the arguments provided to the function. For example, if the argument is provided by a widget that displays text, then the type of the parameter will be `String`. The return type depends on how the function is used. For example, if the function is used as a boolean operation in a constraint, then its return type will be `boolean`. If the function is assigned to a widget that displays text, then the function's return type will be `String`.

   If the operation uses a capability provided by a pre-defined resource, then the body of the corresponding function will contain the statements required to use the capability. Otherwise, the function will have an empty body that needs to be later filled in by the developer. For example, on line 27 in Figure 4, function *savePhone* contains the statement required to create a file in the device's internal storage since the same operation uses *write* capability of the resource *INT_STORE* in the storyboard.

5. A developer can extend the generated definitions of functions. For example, on line 28 in Figure 4, the generated code provides a hook for the developer to extend *savePhone*.

6. A custom resource is translated to an appropriate Android component. The capabilities provided by a custom resource can be accessed via an Android intent. Currently, we only support broadcast receivers as custom resources.

7. The use of a resource in the storyboard indicates an app depends on the resource. Such dependencies are captured in the app's configuration during code generation while relying on the Android system to satisfy these dependencies at runtime in accordance with the device's security policy, e.g., grant permission to use a resource at install time.

4 Implementation of the Methodology

Android JetPack Navigation (AJN) is a suite of libraries that helps Android developers design their apps' navigation in the form of navigation graphs. A navigation graph is a realization of a traditional storyboard in Android Studio. We have extended navigation graphs with features that enable developers to specify an app's storyboard in Android Studio. The developer can visually represent the screens, widgets, and transitions in the navigation graph. While a developer cannot specify operations and constraints visually, she can specify them in the corresponding XML structure of the navigation graph.

   We have extended Android Lint [22], a static analysis tool to analyze files in Android Studio, to implement the analysis and verification of security properties. The analysis is packaged as a Gradle Plugin that can be used from Android Studio.

   We have implemented a code generation tool that takes a navigation graph and translates it into Java code for Android. A developer can extend the generated code with business logic in Java or Kotlin. The code generation tool is also packaged as a Gradle Plugin that can be used from Android Studio.

Designing an app in Android Studio with SeMA   An app developer uses the existing AJN libraries to build a navigation graph of the app. This graph serves as the initial storyboard (e.g., Figure 2). Next, she uses the SeMA provided extensions to the navigation graph to iteratively specify the app's behavior. In each iteration, she uses the extensions to Android Lint tool to analyze the navigation graph for violations of pre-defined security properties. Once the developer has verified the storyboard satisfies the desired properties, she uses the code generation tool to generate an implementation of the app. Finally, she adds the business logic to the generated implementation to complete the implementation of the app.

   All through the development process, the developer operates within the existing development environment while exercising the extensions provided by SeMA.

5 Canonical Examples

We illustrate the methodology with a couple of canonical examples based on vulnerabilities found in real-world apps. Each example demonstrates a vulnerability that can be detected and prevented by the method-
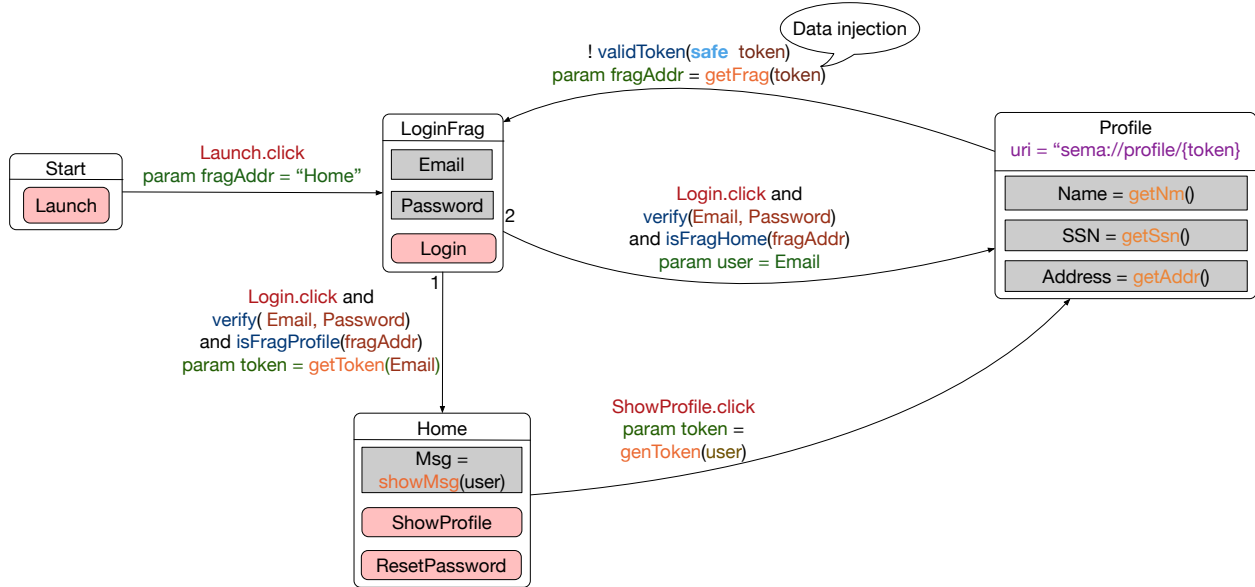
Figure 5: Example illustrating a data injection vulnerability. The (information) bubbles are not part of the storyboard.

ology at design time. For each example, we describe the expected behavior and outline the steps a developer would take to specify the app's behavior in a storyboard and verify the security properties related to confidentiality and integrity. The steps outlined are intended to prescribe one way of developing an app with SeMA. SeMA does not enforce a workflow. Developers may choose to build a storyboard iteratively or they may choose to make it all at once. However, SeMA enforces developers to verify the specified storyboard for security violations before translating it to code.

5.1 Example Illustrating the Use of Information Flow Analysis

Consider an app that allows users to log in and view their profile information. From another app, a valid user of the app can navigate *only* to the screen showing profile information. The example is based on the fragment injection vulnerability discovered in real-world apps [21].

A developer will specify an app in SeMA as follows:

Specify screens and navigation A developer starts by specifying the screens of the app, the widgets in each screen, and the possible transitions between the screens. For example, in Figure 5, a developer initially specifies 4 screens: Start, LoginFrag, Home, and Profile. Each screen has widgets, e.g., the Start screen has one button with label Launch. Finally, the screens are connected to each other via transitions to indicate how the user can navigate between screens, e.g., from the LoginFrag, a user can navigate to either Home or Profile.

Add user action-related constraints to transitions A developer adds constraints to transitions that correspond to actions/gestures performed by users on the widgets in the screens. In Figure 5, such constraints are highlighted in red. For example, the transition from Start to LoginFrag is taken when Launch button in Start is clicked.

Refine transitions with additional constraints A developer adorns a transition with guards/constraints not related to user actions. Such constraints are highlighted in blue in Figure 5. For example, the transition from LoginFrag to Home is taken when the verify operation and the isFragHome operation returns true in addition to the user action-related constraint, that is, Login button clicked. Likewise, the transition from Profile to LoginFrag is taken when the validToken operation returns true.
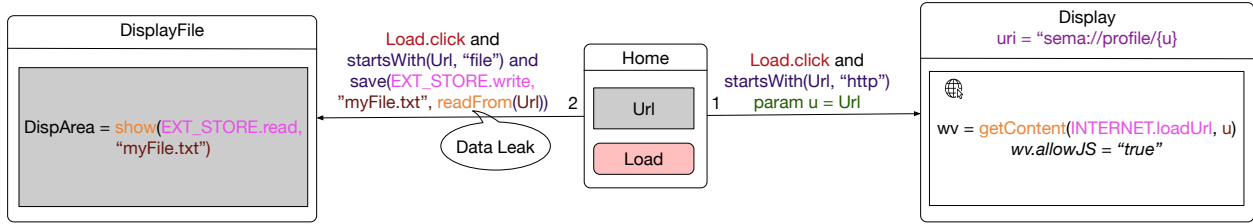
11

Figure 6: Example illustrating a data leak vulnerability. The (information) bubbles are not part of the storyboard.

These boolean operations may have input parameters. For example, the `verify` operation takes two input parameters, the arguments to which are provided by the values in widgets `Email` and `Password`.

Connect input parameters of screens to data sources A developer specifies the data sources that will provide arguments to the input parameters of a screen. These sources can be provided in two ways – (1) as part of incoming transitions to a screen (in green Figure 5) or (2) as part of URIs associated with screens (in purple in Figure 5). For example, the argument to the `token` input parameter of screen `Profile` is provided by a non-boolean operation `getToken` (highlighted in orange in Figure 5) when the transition to `Profile` is taken. On the other hand, the argument of `token` is provided by the `token` variable associated with the URI of screen `Profile` when `Profile` is triggered by an external app via the associated URI.

Check security properties Finally, the developer analyzes the specified storyboard for security property violations, which results in a data injection warning, as shown in Figure 5. This warning is because `isFragProfile` and `isFragHome` operations consume the input parameter `fragAddr` as argument. On the transition from `Profile` screen to `Login Frag` screen, `fragAddr` takes on the return value of `getFrag` operation that consumes `token` parameter of `Profile` screen. Since an external app provides the `token` argument to the `Profile` screen via its URI, an external app can manipulate `token` to gain access to `Home` screen. Information flow analysis will detect and flag this violation by following the chain of flow from untrusted source and sinks.

Apply fix suggested by SeMA The reported vulnerability can be fixed by changing `param fragAddr = getFrag(token)` to `param fragAddr = "profile"` on the transition from `Profile` screen to `LoginFrag` screen as this breaks the dependence between the app's navigation and `token` parameter.

5.2 Example Illustrating the Use of Information Flow Analysis and Rule Checking

Consider an app that allows a user to enter a trusted URL in a text field and displays the content from the URL. If the URL is a file URL, then the data in the file is displayed to the user and downloaded to the device's external storage. The URL can be a file URL or a web URL. The example is based on the sensitive data leak vulnerability discovered in the Firefox app for Android [8] and the Zomato app [15].

A developer will specify the storyboard of this app iteratively as follows:

Specify screens and navigation As in the previous example, a developer starts by specifying the screens, widgets in each screen, and the possible transitions between the screens. As shown in Figure 6, the app has 3 screens – `Home`, `Display`, and `DisplayFile`. The `Display` screen is used to display web content from a URL entered in `Url` in `Home` screen. The `DisplayFile` screen is used to display the contents of a file entered by the user in `Url` in `Home` screen.

Add user action-related constraints to transitions A developer attaches constraints on transitions related to gestures/actions by a user on widgets in a screen (highlighted in red Figure 6). For example,

in the figure, a user can either navigate to `Display` or `DisplayFile` from `Home` upon the click of the `Load` button in `Home`.

Refine transitions with additional constraints A developer adds guards/constraints on transitions not related to user actions (shown in blue Figure 6). These guards are specified as operations (in orange in Figure 6) that return boolean values. For example, the transitions from `Home` to `Display` is taken when the `Load` button is clicked and the `startsWith` operation returns true. The `startsWith` verifies if the URL entered by the user starts with "http:". Likewise, the transition from `Home` to `DisplayFile` is taken when the `Load` button is clicked, the entered URL starts with "file" (see `startsWith` operation in Figure 6), and the `save` operation returns true (i.e., the content in the file path `Url` is saved).

Specify pre-defined resources used by operations A developer adds more detail to the operations via pre-defined resources. For example, a developer specifies the type of local storage that will be used by the `save` operation, as shown in Figure 6 (highlighted in pink). In the storyboard shown, the `save` operation uses the `write` capability provided by the pre-defined resource `EXT_STORE` to write data to a file in the device's external storage.

Connect input parameters of screens to data sources As in the previous example, a developer specifies a data source to `u`, an input parameter of screen `Display` in Figure 6. The source is either the widget `Url` in screen `Home` (in green in Figure 6) or the variable `u` in the URI associated with the `Display` screen (in purple).

Connect widgets to data sources A developer specifies the data source that provides the value to be displayed in a widget. As shown in Figure 6, the values from non-boolean operations `show` and `getContent` are used as values to be displayed in `DispArea` and `wv` widgets respectively. The operation `show` reads a file in the device's external storage and the operation `getContent` loads web content from a URL.

Configure WebView widgets A developer configures the `Webview` widget by setting pre-defined attributes of the widget. For example, in Figure 6 JavaScript execution is allowed by setting the value of the widget's `allowJS` attribute to "true".

Check security properties Finally, a developer analyzes the specified behavior in the storyboard to check for violations related to confidentiality and integrity. The analysis reveals a violation of confidentiality due to a data leak vulnerability. If the file path provided in `Url` is a file in the app's internal storage, then any file from the app's internal storage can be stored in the device's external storage. Thus, all apps installed in the device can access the app's internal files since files in external storage can be accessed by all apps. This violates the confidentiality of the data read from trusted sources. Information flow analysis will detect and flag this violation by following the chain of flow from sources to untrusted sinks. Further, rule checking will flag a violation of Rule 2 because a whitelist of trusted URLs is not specified for `wv` widget via `trust-patterns` attribute. The absence of whitelist will allow `wv` to execute JavaScript embedded in potentially malicious URLs.

Apply fix suggested by SeMA One way to fix this vulnerability is not to save internal files to external storage. Instead, the data in an internal file should be directly displayed in `DisplayFile`. The user should be provided with a button, in `DisplayFile`, to download the displayed data in external storage. This design is more secure than the one described above because data in the app's internal files cannot be exposed without the user's explicit approval via a UI action, e.g., button click. However, SeMA will still report a violation since data is being written to an external storage file. In this case, since the app's design requires explicit consent from the user before writing to external storage, a developer can override this violation by marking the flow of information to the external storage file as *safe*.

The rule violation is fixed by specifying a whitelist of trusted URLs for `wv`, e.g., `wv.trust-patterns=` `{".*sema.org.*"}`.

6   Evaluation

We evaluated SeMA in terms of (1) *Effectiveness* focused on its ability to detect and prevent vulnerabilities, and (2) *Usability* focused on its ability to help developers build apps with features that are often found in real-world apps but without a set of known Android app vulnerabilities.

6.1   Effectiveness

We used the Ghera benchmark suite [33] for this evaluation. Ghera has 60 benchmarks. Each benchmark captures a unique vulnerability. We used Ghera because the vulnerabilities in Ghera benchmarks are *valid*; that is, they have been previously reported in the literature or documented in Android documentation. These vulnerabilities are *general and exploitable* as they can be verified by executing the corresponding benchmarks on vanilla Android devices and emulators. Further, each vulnerability is *current* as they are based on Android API levels 22 thru 27, which enable more than 90% of Android devices in the world and are targeted by both existing and new apps. Finally, the benchmarks are *representative* of real-world apps in terms of the APIs they use, as established by Ranganath and Mitra [38]. Hence, the benchmarks in Ghera are well-suited for this evaluation.

For each Ghera benchmark, we used SeMA to create a storyboard of the *Benign* app. If SeMA detected the vulnerability in *Benign* app's storyboard, then we modified the storyboard till SeMA found no vulnerabilities. We generated the code from this storyboard and verified the absence of the vulnerability captured in *Benign* by executing it with the *Malicious* app. If the *Malicious* app was unable to exploit the generated app, then we concluded that SeMA prevented the vulnerability.

Of the 60 vulnerabilities captured in Ghera benchmarks, SeMA detected 49 vulnerabilities: 15 were detected via information flow analysis, nine were detected via rule checking, six were detected via a combination of info flow analysis and rule checking, and 19 were prevented via code generation. Table 1 provides the breakdown of this information.

Table 1: Results showing how a vulnerability in a benchmark was detected/prevented. CG, IF, and RC refer to Code Generation, Information Flow Analysis, and Rule-based Analysis, respectively.

ID	Benchmark	Method
C1	Block Cipher in ECB mode weakens the encryption.	CG
C2	Non-random IV weakens the encryption.	CG
C3	Use of a constant secret key exposes the secret key.	RC
C4	No authentication of KeyStore exposes the KeyStore.	CG
C5	Constant Salt in password-based secret key exposes the key.	CG
I1	Dynamic registration of broadcast receiver exposes it.	RC
I2	Sharing empty pending intent leads to intent hijacking.	IF
I3	Dynamic Fragment loading enables fragment injection.	IF
I4	Low priority activity can be hijacked by high priority activity.	IF
I5	Sharing implicit pending intent leads to intent hijacking.	IF
I7	Not validating implicit intents grants unauthorized access.	IF
I8	Not verifying a broadcast message exposes the receiver.	RC
I9	Accepting input from a receiver enables data injection.	IF
I10	Not protecting broadcast receiver enables privilege escalation.	RC
I11	Starting activity in a new task enables activity hijack.	CG
I12	Non-empty task affinity of launcher activity enables phishing.	CG
I13	Starting activity in a new task enables phishing.	CG
I14	Non-empty task affinity of an activity enables phishing and DoS.	CG
N1	Incorrect checking of server certificates enables MITM.	CG
N2	Incorrect verification of server hostname enables MITM.	CG
N3	Not verifying the hostname of an SSL socket enables MITM.	CG
N4	*SSLCertificateSocketFactory.getInsecure()* API enables MITM.	CG

Continued on next page

Table 1 – *Continued from previous page*

ID	Benchmark	Method
N5	Incorrect checking of certificate authority enables MITM.	CG
N6	Sending data via open socket enables data theft.	IF
N7	Reading data from an open socket enables data injection.	IF
N8	Absence of pinned certificates enables MITM.	RC
P1	Using unnecessary permissions enables privilege escalation.	CG
P2	Weak permissions do not provide adequate protection.	RC
S1	Reading from files in external storage enables data injection.	IF
S2	Writing to files in external storage enables data theft.	IF
S3	Unsanitized paths to internal storage exposes internal files.	IF
S4	Writing to files in external storage from files in internal storage enables data theft.	IF
Y1	API *CheckCallingOrSelfPermission* does not provide protection.	CG
Y2	API *CheckPermission* does not provide protection.	CG
Y3	API *EnforceCallingOrSelfPermission* does not provide protection.	CG
Y4	API *EnforceCallingOrSelfPermission* does not provide protection.	CG
Y5	Writing data to the clip board enables data theft.	IF
Y6	Executing code from unverified source enables code onjection.	IF
Y7	Sharing unique system ID enable ID theft.	IF
W1	Malicious URLs in a WebView overwrite cookies.	IF & RC
W2	Connecting to HTTP remote servers enables MITM.	RC
W3	Malicious URL in a WebView can inject & execute JS.	IF & RC
W4	Malicious URL in a WebView can embed intents to steal data.	IF & RC
W5	Malicious URL in a WebView can access internal content providers.	IF & RC
W6	Malicious URL in a WebView can access internal files.	IF & RC
W7	Ignoring SSL errors enables MITM.	CG
W8	A WebView not validating resource requests enables MITM.	RC
W9	A WebView with no base URL enables file access to a malicious URL.	IF & RC
W10	A WebView not validating page requests enables MITM.	RC

Two of the vulnerabilities detected via rule analysis could have been detected by code generation. These two vulnerabilities relate to connecting to HTTP remote servers and connecting to HTTPS remote servers without certificate pinning. While *such vulnerabilities can be prevented by code generation, our realization of the methodology relies on rule checking to offer flexibility in using HTTP vs. HTTPS and certificate pinning in storyboards when connecting to remote servers.*

The current realization of SeMA was not applicable to 11 benchmarks in Ghera. Of these, three benchmarks capture vulnerabilities that cannot be prevented by the methodology, e.g., unhandled exceptions. The remaining eight apps use features that are not yet supported by the methodology, e.g., Content Providers.

6.1.1 Comparison with Existing Efforts

Of the 49 vulnerabilities prevented by the methodology, 28 can be detected curatively by source code analysis after implementing the apps. *Detecting the remaining 21 vulnerabilities by source code analysis is harder* due to combinations of factors such as the semantics of general-purpose programming languages (e.g., Java), security-related specifications provided by the developer (e.g., source/sink APIs), and the behavior of the underlying system (e.g., Android libraries and runtime). This observation is supported by Ranganth and Mitra [38], who found that existing source code analysis tools are not effective in detecting vulnerabilities in an earlier version of Ghera which included 15 of these 21 vulnerabilities. Further, Pauck et al. [36] evaluated six prominent static taint analysis tools aimed to detect data leak vulnerabilities in Android apps and discovered that most tools detect approximately 60% of the vulnerabilities captured in the DroidBench 3.0 benchmark suite. Finally, Luo et al. [31] qualitatively analyzed Android app static taint analysis tools and observed that these tools need to be carefully configured (e.g., relevant source/sink APIs) and should

consider application context to accurately detect vulnerabilities in Android apps.

Ranganath and Mitra [38] found that 14 security analysis tools in isolation could detect at most 15 vulnerabilities, and the full set of tools collectively detected 30 vulnerabilities. This result suggests combining different analysis (e.g.,, deep analyses like pointer analysis, shallow analyses like pattern matching) will likely be more effective in detecting vulnerabilities. Our experience with SeMA suggests *the same is likely true in the context of preventing vulnerabilities: a combination of information flow analysis (deep analysis), rule checking (shallow analysis), and code generation (shallow analysis) helped detect and prevent 49 vulnerabilities.*

Gadient et al. [11] found that real-world apps often expose credentials (e.g., crypto keys) in the source code, use insecure communication channels (e.g., HTTP), and use malicious input to load URLs in a WebView. These vulnerabilities are also captured in Ghera and were prevented by SeMA in this evaluation. This finding suggests *there are non-trivial opportunities for techniques like SeMA to help improve security of real-world apps.*

6.2 Usability

While the evaluation with the Ghera benchmarks shows that SeMA can be used to prevent known vulnerabilities in small apps, it does not tell us if SeMA can be used by developers to uncover vulnerabilities in apps with real-world capabilities and features. Further, SeMA extends storyboards, an existing design artifact, to enable formal reasoning of security properties at design time. While formal reasoning approaches have been proven to be effective in terms of uncovering defects in software (e.g., bugs and vulnerabilities), they can be a burden on developers in terms of time to learn and use [42]. Consequently, the adoption of such approaches in domains like mobile app development may be limited.

To address the concerns identified above, we conducted a usability study of SeMA with ten developers and 13 real-world apps. In this study, we will answer the following research questions:

- **RQ1**: Does SeMA affect app development time?

- **RQ2**: Does software development experience affect development time while using SeMA?

- **RQ3**: Does security-related feature development experience affect development time while using SeMA?

- **RQ4**: Do features used in an app affect development time while using SeMA?

- **RQ5**: Does SeMA detect specific (expected) vulnerabilities introduced in an app's storyboard?

- **RQ6**: Does the use of SeMA (instead of the usual Android app development process) increase the likelihood of introducing expected vulnerabilities?

- **RQ7**: Does software development experience affect the vulnerabilities introduced while using SeMA?

- **RQ8**: Does security-related feature development experience affect the vulnerabilities introduced while using SeMA?

In the following sections, we will describe the different aspects of the experiment, explain the various design decisions, and present the results of our analysis on the observed data.

6.2.1 Study Design

We designed an experiment to gain insights into the effect of introducing SeMA to the Android app development process. Specifically, we wanted to determine if SeMA helps a developer prevent vulnerabilities when developing an app. Additionally, we wanted to find out the cost of using SeMA in terms of time taken to build an app with SeMA. Hence, we carried out the following tasks to accomplish the study:

1. Identify real-world Android apps with expected vulnerabilities.

2. Hire developers to participate in the study.

3. Conduct interventions for the developers participating in the study.

4. Create development exercises for the developers participating in the study.

5. Assign exercises to developers.

6. Collect observational data during exercises.

7. Collect information about developer's prior experience and developer's opinion about SeMA.

8. Analyze collected data.

In the following paragraphs, we will explain the details and the rationale for performing the above tasks.

Identifying Real-world Apps: For this experiment, we selected 30 Ghera benchmarks. The selection was guided by the possibility of detecting the captured vulnerabilities via information flow analysis or rule checking. These vulnerabilities are representative of vulnerabilities found in real-world apps based on API usage information, as shown by Ranganath and Mitra [38]. Hence, they are appropriate for this exercise.

Next, we randomly collected 50 apps from Google Play, Android's official app store. We manually analyzed the source code of the apps to look for features used in the 30 Ghera benchmarks. The first 13 apps we analyzed accounted for the features used by 26 of the 30 selected Ghera benchmarks. None of the remaining 37 apps used features used by the remaining four of the selected Ghera benchmarks. Of the 13 apps, seven apps had been publicly reported to have at least one of the 30 Ghera vulnerabilities [16]. Based on the used features, each of the 13 apps were associated with at least two, on average six, and at most 12 vulnerability benchmarks. The associated number of benchmarks hints at the number of vulnerabilities to expect if these apps were recreated. We refer to these vulnerabilities as expected vulnerabilities from hereon. Table 2 provides a brief description of each app and lists the expected vulnerabilities in each app.

Hiring Developers: Since the purpose of this evaluation is to measure the effect of SeMA on Android app development, the obvious candidates for this study are Android app developers. Due to limited local population of developers familiar with Android app development, we reached out to local developers with some professional software development experience. We curated a list of 30 professional developers sourced from personal contacts and sent out personalized emails to each one of them, asking them about their interest in participating in the study. In these emails, we promised to provide a small financial incentive (e.g., gift card) if they participated in the study. Of the 30 developers, 15 agreed to participate in the study. While we started the study with 15 participants, five participants withdrew from the study in the middle. So, we completed the study with ten participants.

The participants have an average of 20 months of software development experience with a minimum of three months and a maximum of 53 months, as shown in Table 3. Seven of them have experience in developing web applications professionally; that is, outside of learning environments such as classrooms or boot camps, and four of the seven participants experienced in web development have 12 months of experience or more. Further, four of ten participants have experience in developing security features (e.g., authentication) for applications. Finally, in terms of programming languages, eight of ten participants use Java frequently, followed by JavaScript and Python, both of which are used by four participants. However, only three participants have experience developing mobile apps professionally, but none for Android. Hence, *participants selected in this study are not expert Android app developers.*

In general, professional developers are likely to learn quickly since many of them have to learn new concepts, tools, and languages on the job. Also, since commercial software often has to meet security standards, such developers have a better understanding of security-related features than beginners. Further, a non-trivial part of Android app development is done in Java. Hence, despite the lack of exposure to Android app development, *the selected group of participants are representative proxies for Android app developers having experience developing security-related features and can serve as good subjects in this evaluation.*

ID	App Name	App Description	Expected Ghera Vuln.	Total
1	IRS2Go	Registered users can check tax refund status and make tax-related payments	I4,N8,S1,S2,W2	5
2	Geico	Registered users can view and download their auto insurance policies	N8,S1,S2,Y7,W2	5
3	Slack	Registered users can chat and call other registered users	I3,I4,I7,N6,N7,N8,W2	7
4	Ancestry	Registered users can track and save their family history details	N8,S2,W2	3
5	MyBlockHR	Registered users can upload tax-related documents and estimate their annual tax	I3,I4,I7,N8,S1,S2,S3,W2	8
6	AESCrypto	Users can encrypt and decrypt messages with a password	C3,S1	2
7	Grab	Users can book transportation and accommodation at a location	W1,W3,W4,W6,W8,W9,W10	7
8	Zomato	Users can order food from restaurants near their location	I7,W1,W3,W4,W6,W8, W9,W10	8
9	Clipboard Manager	Users can take notes and share them via the clipboard or save to a remote server	N8,Y5,W2	3
10	IRCCloud	Users can upload files in their device to the app	I7,S1,S2,S3,S4	5
11	Harvest	Users can upload bills and receipts of their monthly expenditure	I3,I4,S1,S2,S3,S4	6
12	Firefox	Users can view web or file content in a custom browser	S2,S4,W1,W3,W4,W6,W8, W9,W10	9
13	Yandex	Users can view web content in a custom browser and report a browser crash	I1,I8,I9,I10,P2,W1,W3,W4, W6,W8,W9,W10	12

Table 2: A description of the 13 apps along with the expected Ghera vulnerabilities in each app.

Participant	Software	Web	Mobile	Android	Security
404	53	17	0	0	17
704	3	3	0	0	0
1004	10	0	8	0	0
1503	14	14	0	0	14
1803	30	18	6	0	12
1904	6	6	0	0	0
2103	12	0	0	0	0
2403	3	0	2	0	0
2703	48	12	0	0	10
3103	6	3	0	0	0

Table 3: Participant's development experience (in months) in different platforms and with security features.

Conducting Interventions: Since the participants in the study did not know how to make an Android app, we had to train them in Android app development. Hence, we invited all participants to a single 8-hour session. In this session, we introduced them to the fundamental aspects of an Android app (e.g., activity, intents). This introduction included a presentation of the necessary concepts, secure coding guidelines, and a live demonstration of how to develop an Android app using Android Studio. After this session, the participants were given Android-related resources such as the Android documentation and free video tutorials (e.g., Marakana Android tutorials [32]) to learn more about the APIs needed to build Android apps. They were given five days to examine the additional materials independently. After this brief study period, we conducted a group Q&A session with all the participants. This session comprised two parts. The first part of the session was aimed at helping the participants address any confusion they had about developing an app in Android. The second part of the session was meant to provide them with hands-on experience of Android Studio and Android app development in general.

We designed a similar schedule to train the participants in developing Android apps with SeMA. All the participants were invited to a single 8-hour group session. In this session, we introduced the features in SeMA and demonstrated how to build an app with those features. We provided the participants with the documentation to SeMA and five days to peruse the materials. Finally, we conducted a group Q&A session with all participants to help them learn about developing an app with SeMA. In addition to the Q&A, in this session, they made an app with SeMA with our assistance. The hands-on experience with SeMA helped them get acclimatized to Android app development with SeMA.

This entire process of teaching ten participants Android and SeMA, took approximately two weeks to complete.

In any usability study that requires interventions, the application of the interventions to the users should be designed carefully. In this study, since the training received by a participant could affect the results, *it was necessary to ensure that they received homogeneous instruction.* Hence, we conducted all the training and Q&A sessions for the participants together instead of individually. However, since each participant was allowed to independently explore Android and SeMA, the effect of individual learning will affect the results of this experiment.

Creating Development Exercises: We analyzed the selected 13 real-world apps in two ways to design the development exercises for the participants. First, we decompiled the app and manually analyzed its metadata and source code. This analysis revealed the app's screens, any external apps (e.g., remote server) the app communicates with, the data flow of the app, and the APIs used in the apps. Second, we installed the app on an Android device and interacted with it to understand its navigation. Based on the analysis of an app's innards and navigation, we constructed a specification of the app. These specifications were used as exercises for the participants in the study.

The motivation behind using these specifications as exercises for participants was that we wanted to mimic a real-world scenario where a developer would have to start from a description of an app's expected behavior, understand it, and use the relevant features in Android to realize the app. Further, since one of our goals was to measure the ability of SeMA to help developers uncover vulnerabilities, the specifications provided in the exercises were constructed to force participants to make decisions that would affect the app's security.

As an example, consider the snippet from an exercise assigned to a participant in this experiment – *"When the app is started, the user is shown a FileUploader screen. The FileUploader screen has a button to allow users to upload a file. Upon clicking this button, the user is shown a list of files. Upon selecting a file, the selected file is saved, and the user is shown a message to indicate if the file was saved successfully."* This specification asks the participant to design an app that allows users to upload a file, a feature that requires the participant to consider security implications. For example, if the participant decided to use an external client to help the user select a file, then she would register the app to receive the chosen file path from the external client via a message. In doing so, an app might open itself to accepting malicious messages. If the app used the file path embedded in a malicious message to create a destination where the selected file will be uploaded, then this decision would enable data injection or directory traversal attack. Hence, the participant's approach to design the file upload feature will impact the app's security. These decisions and SeMA's influence on them was the focus of this study.

Assigning Exercises to Participants: We assigned three distinct apps to every participant. Of the three apps assigned to each participant, one baseline app was common to all participants (App ID 1 in Table 2). All participants developed the baseline app first with SeMA and then without SeMA; that is, using the existing Android development process. We ensured that the baseline app with SeMA and without SeMA was developed on different days to reduce the effect of memorization. The rationale for developing the baseline app with SeMA before developing the same app without SeMA was to avoid the inflation of SeMA's effect due to a participant's prior knowledge about the app/exercise.

In addition to the baseline app, each participant built two apps using SeMA. This assignment strategy resulted in 40 sample implementations of 13 apps. Of the 40 samples, 20 were realizations of the baseline app – 10 using SeMA and 10 without SeMA. Keeping the baseline app constant across all participants was necessary to measure the effect of SeMA on one app for all participants. The remaining 20 samples were uniformly distributed across the remaining 12 apps (see Table 4). This distribution ensured all expected vulnerabilities could be introduced by a participant.

Observing Development and Collecting Data: Each participant worked on the assigned development exercise via remote sessions. Participants decided on the length of each session. There was no limit on the number of sessions to work on a development exercise.

In every session, participants shared their screen with us. Via screen sharing, we observed the development of apps and recorded when participants introduced vulnerabilities that were expected in the developed apps. Also, we recorded the error messages and causes of error reported by SeMA.

Designing and Administering Development Experience Surveys: We administered two surveys after the participants finished their exercises. The purpose of the first survey was to build a participant profile. It asked about the time spent developing software outside formal learning environments (e.g., classrooms), experience developing web and mobile apps, and familiarity with different programming languages and technologies. The purpose of the second survey was to assess their experience with SeMA. It asked about various features of SeMA and if they aided/impeded the development workflow.

Analyzing Observational Data: To answer RQ1, we analyzed the development time of developing the baseline app with and without SeMA. The development of baseline apps without SeMA served as the control group, while the development of baseline apps with SeMA served as the treatment group. We computed the average difference in app development time between the control and treatment groups and used a paired two-tailed t-test to measure the effect of SeMA on app development time.

To answer RQ2, we partitioned the participants into two groups. The +2DX group comprised participants with two or more years of software development experience and the -2DX group comprised participants with less than two years of development experience. We used a two sample two-tailed t-test to compare the average development time using SeMA between these groups to quantify the effect of software development experience on developing software using SeMA.

To answer RQ3, similar to RQ2, we partitioned the participants into two groups. +1SFDX group comprised of participants with one or more years of security feature development experience and -1SFDX group comprised of participants with less than one year of security feature development experience.

To answer RQ4, we compared the average time taken by participants using SeMA to make apps with expected vulnerabilities stemming from the features of a category and apps without the same features. We used a two sample two-tailed t-test for this comparison.

To answer RQ5, we measured the proportion of total number of expected vulnerabilities introduced by all participants while using SeMA that were then successfully detected by SeMA. We also considered the proportion of expected vulnerabilities that were prevented by SeMA (due to good defaults).

While SeMA is effective in detecting expected vulnerabilities, we need to assess the effect of changing the development methodology on the introduction of expected vulnerabilities. We tackle this concern in RQ6 — *does the use of SeMA (instead of the usual Android app development process) increase the likelihood of introducing expected vulnerabilities?*

To answer RQ6, we measured the average proportion of expected vulnerabilities introduced by participants in the baseline app with and without SeMA. We compared the average proportions with a paired

| | AppID | | | | | | | | | | | |
Participant	2	3	4	5	6	7	8	9	10	11	12	13
404			102 (3)			85 (2)						
704								105 (2)		135(3)		
1004											55 (2)	
1503	150 (2)											
1803				240 (3)	180 (3)			90 (2)				
1904		150 (2)										110 (3)
2103					105 (3)				150 (2)			
2403		120 (2)									60 (3)	
2703							105 (2)			132 (3)		
3103									100 (3)			80 (2)

Table 4: Time taken (in minutes) by a participant to make non-baseline apps. The number in brackets indicates if an app was a participant's second or third app.

two-tailed t-test to determine if there was any significant difference in the average proportion of vulnerabilities introduced in the baseline app with and without SeMA.

To answer RQ7, we computed the proportion of expected vulnerabilities introduced while using SeMA for both +2DX and -2DX groups. We compared these two proportions using a two sample two-tailed z-test to measure the effect of the participants' software development experience on introducing vulnerabilities with SeMA.

To answer RQ8, we compared the proportion of expected vulnerabilities introduced while using SeMA by participants in +1SFDX and -1SFDX group. We used a two sample two-tailed z-test for the comparison.

6.2.2 Results from Observational Data

RQ1: Does SeMA affect app development time? Every participant took less time to make the baseline app with SeMA compared to without SeMA, as shown in columns 2 and 3 of Table 5. The median development time of an app with SeMA is 188 minutes and without SeMA is 283 minutes, which shows that, for the participants considered in this study, *developing the baseline app with SeMA took 40% less than developing the same app without SeMA.*

Based on paired two-tailed t-test, SeMA has a significant effect (p-value = 0.0002 with 95% confidence interval [79 mins., 177 mins.]]) on the mean development time in developing the baseline app. The 95% confidence interval of SeMA's effect on decreasing the development time suggests that SeMA has the potential to reduce app development time.

RQ2: Does software development experience affect development time while using SeMA? The three participants in the +2DX group took 142 minutes on average, to develop an app with SeMA. This time was 150 minutes for the seven participants in the -2DX group. Based on two sample two-tailed t-test comparing the means of these two groups, there is no significant difference in the average time taken by participants from these two groups while using SeMA (p-value = 0.75 with 95% confidence interval [-55 mins., 39 mins.]). Hence, *software development experience (not related to Android app development) does not affect the time taken by a developer to make an app with SeMA.*

RQ3: Does security-related feature development experience affect development time while using SeMA? The four participants in the +1SFDX group took 160 minutes on average, to make an app with SeMA. This time was 139 minutes for the six participants in the -1SFDX group. While participants in the +1SFDX group took more time than participants in the -1SFDX group in this sample, the difference is not significant based on two sample two-tailed t-test (p-value = 0.37 with 95% confidence interval [-28 mins., 70 mins.]). Hence, *security-related feature development experience does not affect the time taken by a developer to make an app with SeMA.*

RQ4: Do the features used in an app affect development time while using SeMA? With SeMA, for the seven apps with ICC features based expected vulnerabilities (ICC apps), the average development time was 58 minutes more than the average development time of the six apps without ICC-based expected vulnerabilities (non-ICC apps). Based on two sample two-tailed t-test, the 95% confidence interval of the difference between the average development time to make ICC apps and non-ICC apps ranges from 21 to 95 minutes. A likely reason for longer development time for ICC-apps is that ICC-apps tend to have more screens and navigation between the screens. Considering the significant difference in development time for at least one set of features (e.g., ICC), *the features used in an app are likely to affect the time taken to make the app with SeMA.*

Based on the average development time using SeMA, the development of baseline apps took more time than the development of non-baseline apps (see the *average* row in Table 5). The longer development time for the baseline app is likely due to the methodology's novelty, considering the baseline app was the first app participants developed using SeMA. This observation suggests that development time reduced as participants became familiar with SeMA. However, development time did not necessarily decrease consistently across all apps made with SeMA. For example, participant 1803 took more time to develop the third app than the second app. This increase was likely because the third app had more features than the second app, making the third app more complicated. Specifically, the third app was more complicated since it had features related to user registration and authentication, which were absent in the second app. Hence, *the reduction in development time with SeMA seems to be dependent on the familiarity with SeMA and the features used in the app.*

| Participant | Time to Baseline App | | Time to | | Expected Vulns. | Added Vulns. |
| | no SeMA | SeMA | 2nd App | 3rd App | | |
			SeMA			
404	276	165	85	102	15	10
704	440	255	105	135	14	4
1004	180	140	55	135	22	10
1503	588	315	180	150	12	5
1803	360	210	90	240	16	7
1904	290	210	150	110	24	16
2103	390	210	150	105	12	5
2403	210	138	120	60	21	12
2703	240	150	105	132	19	11
3103	240	135	80	100	22	11
Average	321	193	112	138		
95% C.I		————[124,170]————				

Table 5: Observational Data of each participant. The unit of time is minute. Only the baseline app was made with and without SeMA. The non-baseline apps were made only with SeMA. *Expected Vulns.* indicates no. of vulnerabilities that could have been introduced by a participant while developing the three apps assigned to her with SeMA. *Added Vulns.* indicates the no. of vulnerabilities introduced by a participant while developing the three apps assigned to her with SeMA.

Participant	N	S	T	W	K	D
404	3	2	5	0.6	0.4	0.2
704	2	0	5	0.4	0	0.4
1004	2	2	5	0.4	0.4	0
1503	2	3	5	0.4	0.6	-0.2
1803	2	2	5	0.4	0.4	0
1904	2	3	5	0.4	0.4	0
2103	1	1	5	0.2	0.2	0
2403	2	0	5	0.4	0	0.4
2703	2	1	5	0.4	0.2	0.2
3103	2	1	5	0.4	0.2	0.2
Average				0.4	0.3	0.1

Table 6: Proportion of expected vulnerabilities introduced in the baseline app by a participant with and without SeMA. N indicates the no. of expected vulnerabilities introduced by a participant in the baseline app without using SeMA. S indicates the no. of expected vulnerabilities introduced by a participant in the baseline app using SeMA. T indicates the no. of expected vulnerabilities in the baseline app. W indicates the proportion of expected vulnerabilities introduced by a participant in the baseline app without SeMA. K indicates the proportion of expected vulnerabilities introduced by a participant in the baseline app with SeMA. D is the difference between W and K.

RQ5: Does SeMA detect specific (expected) vulnerabilities introduced in an app's storyboard? The participants in this study introduced 91 of 177 instances of expected vulnerabilities in 30 app storyboards while using SeMA (see *Added Vulns.* column in Table 5). For example, participants introduced all instances of expected vulnerabilities in the Web category (see Table 7). SeMA detected and reported a violation for every one of the 91 instances. Hence, *SeMA is highly likely to detect expected vulnerabilities introduced in a storyboard.*

We anticipated SeMA would prevent 38 instances of expected vulnerabilities – two instances of I1, 16 instances of I4, two instances of I10, and 18 instances of N8 – due to good defaults provided by SeMA, e.g., certificate pinning is enabled by default. Since participants overrode the defaults in three instances – two instances of I4 and one instance of I10, 35 of the 38 instances of expected vulnerabilities were prevented by the good defaults in SeMA. Hence, *the defaults built into SeMA are likely to prevent expected vulnerabilities.*

Overall, since SeMA detected and prevented 126 of 177 (71%) instances of expected vulnerabilities in the considered sample, we can conclude *SeMA is likely to detect and prevent expected vulnerabilities in an app's storyboard.*

Participants did not introduce 51 instances of expected vulnerabilities – one in Crypto, five in ICC, 17 in Networking, 27 in Storage, and one in System – while using SeMA. For example, in the Networking category, 15 of 18 instances of W2 were not introduced since participants decided to use HTTPS over HTTP when communicating with a remote server. In Storage category, when participants introduced S2 (writing to external storage), SeMA flagged the vulnerability. Since this informed the participants about the pitfalls of using external storage, they were more cautious with the further use of external storage. Consequently, only 5 of the 19 expected instances of S1 (reading from external storage) vulnerability were introduced by the participants. Hence, *prior knowledge of vulnerabilities is likely to affect the introduction of vulnerabilities and, consequently, the lower the observed effectiveness of SeMA.* On the positive side, unlike passive interventions like lists of vulnerabilities, *SeMA is likely to serve as an active intervention and improve developer awareness about vulnerabilities.*

RQ6: Does the use of SeMA (instead of the usual Android app development process) increase the likelihood of introducing expected vulnerabilities? The average percentage of introducing an expected vulnerability in the baseline app without SeMA was 40% across all participants (column W in Table 6). With SeMA, the average percentage of introducing an expected vulnerability in the baseline app was 30% (column K in Table 6). The median and mean difference between the percentage of expected vulnerabilities introduced with and without SeMA in the baseline app for each participant is 0% and 10%, respectively (see column D of Table 6). Further, based on paired two-tailed t-test, there is no significant difference between the mean percentage of introducing an expected vulnerability in the baseline app with and without SeMA (p-value = 0.17 with 95% confidence interval [-6%, 26%]). Consequently, *SeMA is not likely to introduce any more or less vulnerabilities compared to the prevalent app development process.*

RQ7: Does software development experience affect the vulnerabilities introduced while using SeMA? If we consider the participants' software development experience, then the three participants in the +2DX group introduced 53% of 50 expected vulnerabilities while using SeMA. The seven participants in the -2DX group introduced 50% of 127 expected vulnerabilities while using SeMA. The difference in the percentage of introducing an expected vulnerability between the two groups of participants is not statistically significant (p-value = 0.44 with 95% confidence interval [-10%, 22%]). Hence, *software development experience does not affect the proportion of expected vulnerabilities introduced by a developer in an app while using SeMA.*

RQ8: Does security-related feature development experience affect the vulnerabilities introduced while using SeMA? The four participants with one year or more experience in security-related feature development introduced 54% of 62 expected vulnerabilities while using SeMA. The six participants with less than a year's experience introduced 49% of 115 expected vulnerabilities while using SeMA. Further, there was no significant difference between the proportion of expected vulnerabilities introduced by the two groups of participants (p-value = 0.72 with 95% confidence interval [-13%, 17%]). A possible reason for the absence of a statistically significant difference is, while the participants in the +1SFDX group have prior knowledge of security features, they do not have experience in security feature development in the context of Android apps. Hence, *security-related feature development experience not specific to Android does not affect the proportion of expected vulnerabilities introduced by a developer in an app while using SeMA.*

Category	Benchmark ID	# Apps (A)	# Expected (E)	# Introduced (I)
Crypto	C3	1	2	1
ICC	I1	1	2	0
	I3	2	4	3
	I4	4	16	2
	I7	5	9	9
	I8	1	2	0
	I9	1	2	0
	I10	1	2	1
ICC Ave.		2.1	5.3	2.1
Networking	N6	1	2	1
	N7	1	2	1
	N8	6	18	0
	W2	7	18	3
Networking Ave.		3.8	10	1.25
P2	Permission	1	2	2
Storage	S1	6	19	5
	S2	7	20	10
	S3	3	6	5
	S4	3	6	4
Storage Ave.		4.8	12.8	6
System	Y5	1	2	2
	Y7	1	1	0
System Ave.		1	1.5	1
Web	W1	4	6	6
	W3	4	6	6
	W4	4	6	6
	W6	4	6	6
	W8	4	6	6
	W9	4	6	6
	W10	4	6	6
Web Ave.		4	6	6
Total Ave.		3.1	6.8	3.5

Table 7: Frequency distribution of each Ghera vulnerability. A indicates no. of apps with an expected vulnerability. E indicates no. of times a vulnerability was expected to be introduced. I indicates no. of times an expected vulnerability was introduced and prevented.

6.2.3 Observations from Developement Experience Survey Data

As mentioned previously, we conducted a survey to collect responses from participants about their experience using SeMA. In the following paragraphs, we present our observations based on a qualitative analysis of the participants' responses. The participant responses are summarized in Table 8

Did the SeMA extension to Android Studio help/impede in Android app development? All participants found the extensions to Android Studio helpful. These extensions included the extensions added to a storyboard (described in Section 3) and capabilities related to generating code from a storyboard. Seven of the ten participants said that the extensions helped them *a lot*. The remaining three said that the extensions aided their workflow *quite a bit*. On the flip side, all participants, except one, said that the extensions did not impede their development. Even the one exception said that the extensions only impeded *a little*. Hence, *the participants in this study found the SeMA extension to Android Studio to be largely useful.*

Did the property annotation feature of SeMA help/impede in Android app development? Eight of the ten participants found that the property annotations in a storyboard (e.g., constraints on transitions) helped them in Android app development *a lot*. The other two felt that the annotations helped them *quite a bit*. Only one of the ten participants said that the annotations impeded her development *a little*. *The participant responses show that the property annotation feature largely aided in app development and did not impede their workflow.*

Did the property checking feature of SeMA help/impede in Android app development? All the participants, except one, said that the property checking feature in SeMA helped them *a lot*. The one exception said that it helped her *quite a bit*. However, five participants said that it impeded their development *a little* and *quite a bit* while the remaining five said that the property checking feature did not impede at all. Since the property checking feature in SeMA forces developers to fix property violations before moving to the next step, a few participants found it impeding their workflow. Hence, *while property checking was helpful for most participants; there is room for improving the process of reporting and fixing violations.*

Did the pre-defined properties provided by SeMA help/impede in Android app development? Six of the ten participants felt that the pre-defined properties (e.g., resources) helped their development *a lot*, and four of them felt that they helped *quite a bit*. In a similar vein, six of them found that the pre-defined properties did not impede their workflow at all, and four of them said that they impeded *a little*. Hence, *participants felt that the pre-defined properties largely aided their workflow.* The four participants who found the pre-defined properties to be *a little* invasive said that it was because they had to keep referring to the documentation for understanding how to use them. This problem can be addressed by adding more auto-completion support for SeMA.

Feedback: What would you change in SeMA? All the participants agreed that storyboard-driven development helped them in their development. They particularly appreciated the visualization of the storyboard as it helped them conceptualize the app's behavior. Further, most of them said that SeMA helped them uncover vulnerabilities in the app's design that they would not have otherwise discovered. Typical feedback from participants was to improve the code-completion aid for SeMA and provide access to documentation in the IDE.

6.2.4 Threats to Validity

While the results from this study provide useful insights about the usability of the SeMA methodology, the small number of participants and the small number of apps made by each participant used in this study might affect the generalize-ability of the results. This limitation can be addressed by repeating the experiment with a large number of participants or increasing the number of apps made by a participant.

Despite the participants in this study having varied experience, the selected participants might not be representative of all kinds of real-world developers. This limitation can be addressed by repeating the study with a more varied sample of developers.

Question	Scale			
	None	A little	Quite a bit	A lot
How much did the SeMA extension to Android Studio *help* in Android app development?	0	0	3	7
How much did the SeMA extension to Android Studio *impede* in Android app development?	9	1	0	0
How much did the property annotation feature of SeMA *help* in Android app development?	0	0	2	8
How much did the property annotation feature of SeMA *impede* in Android app development?	9	1	0	0
How much did the pre-defined properties provided by SeMA *help* in Android app development?	0	0	4	6
How much did the pre-defined properties provided by SeMA *impede* in Android app development?	6	4	0	0
How much did the property checking feature of SeMA *help* in Android app development?	0	0	1	9
How much did the property checking feature of SeMA *impede* in Android app development?	5	5	0	0

Table 8: Questions asked to participants in a survey. The number in each cell indicate the number of participants who chose a particular scale for a question.

While the participants in this study are reasonable proxies for average Android app developers, they did not have Android app development experience outside of learning environments. Hence, the results and observations from this study might change if repeated with participants with experience in Android app development.

The representation of the Ghera vulnerabilities in the sample of real-world apps selected in this study was not uniform; that is, some vulnerabilities appeared more than others. This lack of uniformity could have affected the results. This concern can be addressed by repeating the experiment with a sample of real-world apps containing a more balanced distribution of vulnerabilities.

While the exercises assigned to the participants were based on real-world apps, it is possible that they were influenced by our knowledge of vulnerabilities that occur in Android apps. This influence could have introduced bias for certain vulnerabilities in the exercises. This limitation can be addressed by repeating the experiment with a different set of exercises.

While we ensured that each participant received the same intervention regarding Android app development and SeMA, their personal capacities in grokking new material might have affected the way they made an app. This difference could have influenced the final results.

Finally, we silently observed each participant when they were developing an app. This environment might have caused some participants to behave differently, which might have impacted the way they made an app. The influence of such factors can be verified by conducting studies that consider environment-related aspects.

7 Open Challenges

Performance The current realization and evaluation of SeMA has focused on ensuring the correctness of generated code. So, while SeMA adheres to performance guidelines outlined in Android's documentation in the generated code, the generated code may not be performant. This concern can be verified by evaluating the runtime performance of generated code.

Property preservation One challenge that remains to be addressed in the current realization of SeMA is ensuring the generated implementation satisfies the security properties satisfied by the storyboard. This hinges on ensuring (1) the integrity of generated code and (2) developer-added business logic does not violate

the security properties verified in the app storyboard. The current realization of SeMA deters developers from modifying the generated code. The generated code is kept separate from the developer-added code in Android Studio. If a developer modifies generated code, a warning message is shown to the user as a pop-up. If the developer modifies the generated code in spite of the warning, then the code is re-generated when the developer compiles the app. While these methods discourage the developer from modifying the generated code, they do not prevent it. One way to prevent modifications to generated code is to use techniques (e.g., fingerprinting) to check and enforce the integrity of the generated code.

While SeMA deters the modification of generated code, developers can add business logic code in a way that may not guarantee property preservation. This challenge can be tackled by inhibiting the execution of an app upon detecting the violation of security properties (using techniques such as runtime checks and app sandboxing). However, the current realization of SeMA does not address this concern.

Model modification Updating code after modifying the model, from which the code was derived, is a challenge in MDD [43]. Since SeMA is based on MDD, this challenge applies to SeMA as well. In the current realization of SeMA, when the storyboard is updated, code (including developer-added code) is re-generated. While this approach makes it difficult to update the storyboard, it can be addressed by extending SeMA with a mechanism to identify and retain developer-added code associated with parts of the storyboard that haven't been modified.

8   Artifacts

The current realization of SeMA along with the instructions to build and use it is available in the public repository https://bitbucket.org/secure-it-i/sema/src/master/.

The Ghera benchmarks re-built with SeMA as listed in Table 1 are available in the public repository https://bitbucket.org/secure-it-i/sema/src/master/ghera-apps/.

The raw data collected during the usability study, along with the statistical tests used to analyze the collected data are available in the public repository https://bitbucket.org/secure-it-i/sema/src/master/usability-test/.

9   Related Work

The current prevalent approach to detect vulnerabilities in an Android app is based on analyzing an app's source code [45]. The proposed methodology is different from this approach since it enables the detection of known vulnerabilities at the design stage of the app development process, thereby preventing the creation of vulnerabilities.

Prior research has explored solutions based on MDD to simplify mobile app development [49], reduce technical complexity and development cost [19], and enable the development of cross-platform mobile apps [18, 5, 9]. However, these solutions are not focused on securing mobile apps. Unlike such existing MDD-based approaches to app development, the primary purpose of SeMA is to enable the reasoning of an app's security properties while designing an app.

Most academic efforts related to model-driven development of mobile apps have not explicitly considered security requirements. Few commercial efforts related to model-driven development of mobile apps have considered security requirements limited to specific security aspects, e.g., access control. For instance, the Mendix App Platform allows developers to specify models for app components, define user roles and accesses w.r.t to the app components, and execute the models in a runtime environment to check for access inconsistencies [20]. In comparison, SeMA enables the verification of properties about different security aspects albeit limited to known vulnerabilities.

Most prior MDD efforts in software and mobile app development are based on existing software artifacts, e.g., UML, finite state machines. Basin et al. [30] extended UML to SecureUML to enable the formal specification of systems' access control requirements. Usman et al. [48] relied on use case diagrams, UML class diagrams, and UML state machines in Mobile Apps Generator (MAGS) methodology to capture the specification of an app's requirements, structure, and behavior. Francese et al. [10] used finite state machines as a modeling substrate to capture the data-flow, control-flow, and user interactions in an app. In AXIOM

methodology, Xiaoping et al. [25] used the Abstract Model Tree (AMT) representation to model platform-independent app behavior and requirements. Unlike these solutions, instead of relying on artifacts (e.g., UML models, finite state machines) that are alien to current mobile app development workflows, SeMA is based on storyboards, an essential artifact in the design and development of mobile apps.

Prior research efforts have highlighted the need for considering security at the early stages of the software development process [12, 7, 3]. This need has manifested in the development of *security requirements engineering*: identifying a system's security-related requirements and checking if the system design satisfies the identified requirements. In this spirit, Hayley et al. [17] proposed a framework to define security requirements, context, and assumptions of a system and validate these requirements via formal and informal structured arguments. Haralmbos et al. [34] proposed Secure Tropos, a software development methodology that combines requirements engineering concepts with security-related concepts to aid the design and development of secure software systems. Riaz et al. [39] proposed DIGS framework to help requirements analysts define security goals and verify the completeness and correctness of the requirements w.r.t the defined goals. SeMA is similar to such solutions as it enables specification and verification of security properties of an app's design. However, unlike these solutions, the objective of SeMA is to verify security-related requirements stemming from known vulnerabilities. While this objective limits the security-related requirements that can be tackled with SeMA, it reduces the burden of identifying security-related requirements when using SeMA.

10   Conclusion

In this paper, we explored an alternative (and complementary) approach to develop secure Android apps. This approach focuses on preventing vulnerabilities instead of the traditional approach of detecting vulnerabilities after they have occurred. To this end, SeMA is a design-based methodology based on Model-Driven development and existing design techniques to help build secure Android apps.

SeMA extends storyboards with features that enable developers and designers to collaborate and specify an app's behavior iteratively, while also reasoning about and verifying security properties related to confidentiality and integrity in an app's design. Furthermore, SeMA has code generation support that helps translate annotated/extended storyboards, specified in SeMA, to an implementation. Developers can enrich the generated implementation with business logic code in Java or Kotlin.

A proof-of-concept realization of SeMA is available for Android Studio. An empirical evaluation of SeMA shows that SeMA can prevent 49 of 60 vulnerabilities captured in the Ghera benchmark suite through a combination of information flow analysis, rule checking, and code generation techniques.

A usability study of SeMA with ten professional software developers shows that SeMA is highly likely to reduce the cost of development and prevent known Android app vulnerabilities. The same study showed that the vulnerabilities prevented by SeMA are highly likely to be introduced by developers. Hence, SeMA is not only useful in preventing vulnerabilities, but it is also relevant.

References

[1] ADITYA NARKAR, JOYDEEP MITRA, V.-P. R. Cve-2019-9463, 2019. Available at https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9463/.

[2] APPLE. Storyboards in Xcode. https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html, 2019. Accessed: 05-Feb-2019.

[3] ASSAL, H., AND CHIASSON, S. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)* (2018), USENIX Association, pp. 281–296.

[4] BRAMBILLA, M., CABOT, J., AND WIMMER, M. *Model-Driven Software Engineering in Practice: Second Edition*, 2nd ed. Morgan & Claypool Publishers, USA, 2017.

[5] BRAMBILLA, M., MAURI, A., AND UMUHOZA, E. Extending the interaction flow modeling language (ifml) for model driven development of mobile applications front end. In *Mobile Web Information Systems* (Cham, 2014), I. Awan, M. Younas, X. Franch, and C. Quer, Eds., Springer International Publishing, pp. 176–191.

[6] CHECKMARX. How attackers could hijack your android camera to spy on you, 2019. Available at https://www.checkmarx.com/blog/how-attackers-could-hijack-your-android-camera.

[7] DESIGN, I. C. F. S. Avoiding The Top 10 Software Security Design Flaws. https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/Top-10-Flaws.pdf, 2014. Accessed: 01-Jun-2020.

[8] DONGSONG, Y. Download arbitrary files to sd card via additional slashes in file: Uri, 2014. Available at https://bugzilla.mozilla.org/show_bug.cgi?id=1050690.

[9] FATIMA, I., ANWAR, M. W., AZAM, F., MAQBOOL, B., AND TUFAIL, H. Extending interaction flow modeling language (ifml) for android user interface components. In *Information and Software Technologies* (Cham, 2019), R. Damaševičius and G. Vasiljevienė, Eds., Springer International Publishing, pp. 76–89.

[10] FRANCESE, R., RISI, M., SCANNIELLO, G., AND TORTORA, G. Model-driven development for multi-platform mobile applications. In *Proceedings of the 16th International Conference on Product-Focused Software Process Improvement - Volume 9459* (Berlin, Heidelberg, 2015), Springer-Verlag, p. 61–67.

[11] GADIENT, P., GHAFARI, M., TARNUTZER, M.-A., AND NIERSTRASZ, O. Web apis in android through the lens of security. In *Proc. SANER* (2020), IEEE, pp. 13–22.

[12] GEER, D. Are companies actually using secure development life cycles? *Computer 43*, 6 (2010), 12–16.

[13] GOOGLE. Navigation, 2019. Available at https://developer.android.com/guide/navigation.

[14] GREEN, M., AND SMITH, M. Developers are not the enemy! the need for usable security apis. *IEEE Security and Privacy 14*, 5 (2016), 40–46.

[15] HACKERONE. Theft of user session, 2019. Available at https://hackerone.com/reports/328486.

[16] HACKERONE. Hackerone – hack for good, 2020. Available at https://www.hackerone.com/.

[17] HALEY, C., LANEY, R., MOFFETT, J., AND NUSEIBEH, B. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering 34*, 1 (2008), 133–153.

[18] HEITKOTTER, H., KUCHEN, H., AND MAJCHRZAK, T. A. Extending a model driven cross-platform development approach for business apps. *Science of Computer Programming 97* (2015), 31 – 36. Special Issue on New Ideas and Emerging Results in Understanding Software.

[19] HEMEL, Z., AND VISSER, E. Declaratively programming the mobile web with mobl. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2011), OOPSLA '11, Association for Computing Machinery, p. 695–712.

[20] HENKEL, M., AND STIRNA, J. Pondering on the key functionality of model driven development tools: The case of mendix. In *Perspectives in Business Informatics Research* (2010), Springer Berlin Heidelberg, pp. 146–160.

[21] IBM. Android collapses into fragments, 2014. Available at https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf.

[22] INC., G. Android lint overview, 2020. Available at http://tools.android.com/lint/overview.

[23] INSTITUTE, C. M. S. E. Android secure coding standards, 2014. Available at https://wiki.sei.cmu.edu/confluence/display/android/Android+Secure+Coding+Standard.

[24] JACKSON, D. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11* (2002), 256–290.

[25] JIA, X., AND JONES, C. Cross-platform application development using axiom as an agile model-driven approach. In *Software and Data Technologies* (Berlin, Heidelberg, 2013), J. Cordeiro, S. Hammoudi, and M. van Sinderen, Eds., Springer Berlin Heidelberg, pp. 36–51.

[26] JÜRJENS, J. Umlsec: Extending uml for secure systems development. In *UML 2002 — The Unified Modeling Language* (2002), J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds., Springer Berlin Heidelberg, pp. 412–425.

[27] LAMPORT, L. Who builds a house without drawing blueprints? *Commun. ACM 58*, 4 (Mar. 2015), 38–41.

[28] LI, L., BISSYANDÉ, T. F., PAPADAKIS, M., RASTHOFER, S., BARTEL, A., OCTEAU, D., KLEIN, J., AND TRAON, L. Static analysis of android apps: A systematic literature review. *Information and Software Technology 88* (2017), 67 – 95.

[29] LITTLE, A. Storyboarding in the Software Design Process. https://uxmag.com/articles/storyboarding-in-the-software-design-process, 2013. Accessed: 05-Feb-2019.

[30] LODDERSTEDT, T., BASIN, D., AND DOSER, J. Secureuml: A uml-based modeling language for model-driven security. In *UML 2002 — The Unified Modeling Language* (Berlin, Heidelberg, 2013), Springer Berlin Heidelberg, pp. 36–51.

[31] LUO, L., BODDEN, E., AND SPATH, J. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), IEEE, pp. 102–114.

[32] MARAKANA. Android App Development Tutorials. https://www.youtube.com/playlist?list=PLC6F613CB266444FF, 2020. Accessed: 01-May-2020.

[33] MITRA, J., AND RANGANATH, V.-P. Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering* (2017), ACM, pp. 43–52.

[34] "MOURATIDIS, H., AND GIORGINI, P. "secure tropos: A security-oriented extension of the tropos methodology". In "International Journal of Software Engineering and Knowledge Engineering" ("2007"), "World Scientific", pp. "285–309".

[35] OWASP. Mobile security project, 2017. Available at https://wiki.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Controls.

[36] PAUCK, F., BODDEN, E., AND WEHRHEIM, H. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), ACM, pp. 331–341. https://foellix.github.io/ReproDroid/.

[37] PRUDNIKOV, D. UX Design Techniques for Mobile Apps. https://yalantis.com/blog/ux-design-techniques-mobile-app-design/, 2019. Accessed: 05-Feb-2019.

[38] RANGANATH, V.-P., AND MITRA, J. Are free android app security analysis tools effective in detecting known vulnerabilities? *Empirical Software Engineering* (2019).

[39] RIAZ, M., STALLINGS, J., SINGH, M. P., SLANKAS, J., AND WILLIAMS, L. Digs: A framework for discovering goals for security requirements engineering. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2016), Association for Computing Machinery.

[40] RICHTERS, M., AND GOGOLLA, M. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language* (Berlin, Heidelberg, 2002), Springer-Verlag, p. 42–68.

[41] SADEGHI, A., BAGHERI, H., GARCIA, J., AND MALEK, S. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering 43*, 6 (June 2017), 492–530.

[42] SCHAFFER, K., AND VOAS, J. What happened to formal methods for security? *IEEE International Conference on Software Maintenance (ICSM) 49*, 8 (2016), 70–79.

[43] SELIC, B. What will it take? a view on adoption of model-based methods in practice. *Softw. Syst. Model. 11*, 4 (2012), 513–526.

[44] SKETCH. Sketch Design ToolKit. https://www.sketchapp.com/, 2019. Accessed: 05-Feb-2019.

[45] SUFATRIO, TAN, D. J. J., CHUA, T.-W., AND THING, V. L. L. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.* (2015), 58:1–58:45.

[46] TECHNOLOGIES, P. Vulnerabilities and threats in mobile applications, 2019, 2019. Available at https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/.

[47] TRAPP, M., AND YASMIN, R. Addressing animated transitions already in mobile app storyboards. In *Design, User Experience, and Usability. Web, Mobile, and Product Design* (2013), A. Marcus, Ed., Springer Berlin Heidelberg, pp. 723–732.

[48] USMAN, M., IQBAL, M. Z., AND KHAN, M. U. A model-driven approach to generate mobile applications for multiple platforms. In *2014 21st Asia-Pacific Software Engineering Conference* (2014), vol. 1, pp. 111–118.

[49] VAUPEL, S., TAENTZER, G., GERLACH, R., AND GUCKERT, M. Model driven development of platform-independent mobile applications supporting role-based app variability. In *Software Engineering 2016* (2016), Gesellschaft für Informatik e.V., pp. 99–100.

[50] WANG, Y., LIU, X., MAO, W., AND WANG, W. Dcdroid: Automated detection of ssl/tls certificate verification vulnerabilities in android apps. In *Proceedings of the ACM Turing Celebration Conference - China* (2019), ACM TURC '19, Association for Computing Machinery.

[51] WAYNE, H. *Practical TLA+: Planning Driven Development*, 1st ed. Apress, USA, 2018.

A   Formal Syntax of SeMA

Before specifying the syntax of the storyboard language, we first list the sets needed to specify the syntax. Boolean Values BV = {*true*, *false*}, non-boolean values V, Apps APP, Screens S, Proxy screens PS, App Identifiers APPID, Screen Identifiers SID, URIs U, URI strings US, Widgets W, Widget Identifiers WID, Widget Types WT, Gestures on widgets G, User actions UA, Transitions TR, Transition Identifiers TID, Boolean expressions B, Resources R, Resource Names RN, Capabilities provided by resources C, Capability Names CN, argument to operations ARG, parameter names of screens VID, Operations F, Operation Names FN, and Input Parameters of screens P.

Assume that the syntactic structure of values, identifiers, names, and URI strings is given. For example, identifiers consist of non-empty string of letters. The remaining syntactic sets are defined inductively via formation rules shown below.

The formation rules use the following meta-variables to range over the syntactic sets:

bv ranges over BV, v ranges over V, *app* ranges over APP, *appid* ranges over APPID, *vid* ranges over VID, *wid* ranges over WID, *sid* ranges over SID, a ranges over ARG, s ranges over S, *ps* ranges over PS, w ranges over W, *wt* ranges over WT, *tr* ranges over TR, b ranges over B, r ranges over R, *rn* ranges over RN, f ranges over F, p ranges over P, *fn* ranges over FN, *tid* ranges over TID, u ranges over U, *us* ranges over US, c ranges over C, *cn* ranges over CN, and g ranges over G.

The meta-variables can be sub-scripted or primed e.g. s', s_0 stands for an element in the set S. The symbol ? is used to denote an *optional* syntactic construct. The formation rules are presented in a variant of BNF (Backus-Naur form) in Figure 7.

app ::= $appId$ s r?

s ::= **screen** sid u? w tr? | **proxy safe**? sid $appId$? us

w ::= **safe**? wt wid $(v \mid f \mid vid) \mid w_0 w_1$

wt ::= **TextView** | **EditText** | **Button** | **WebView**

f ::= **fun** fn $(rn.cn)$? a?

r ::= (**access all** | **user** | **own**) resource rn c | $r_0 r_1$

c ::= **priv**? cn f | $c_0 c_1$

a ::= (**safe**? $v \mid vid \mid wid \mid$ f) | $a_0 a_1$

tr ::= **transition** tid **dest** sid **cond** ua (**and** b)? p? | $tr_0 tr_1$

ua ::= wid.g

g ::= **click** | **swipe** | **drag**

p ::= **param** vid $(wid \mid vid_0 \mid v \mid$ f) | $p_0 p_1$

u ::= $us \mid us/k \mid u_0 u_1$

k ::= $vid \mid k_0 k_1$

b ::= $bv \mid$ f $\mid b_0$ **and** $b_1 \mid b_0$ **or** $b_1 \mid$ **not** b

Figure 7: Formation Rules

B Formal Semantics of SeMA

Before describing the small-step operational semantics of a storyboard in SeMA, we introduce the function and operators in the meta-language that will be needed to understand the semantics.

1. $\phi : ID \to SID$, where $ID = VID \cup WID$ is a function that is used to keep track of the screen associated with a given widget or variable identifier. For a widget identifier $wid \in WID$, it returns a screen identifier sid where wid is a widget in sid. For a variable identifier $vid \in VID$, it returns a screen identifier sid such that vid as an input parameter of sid.

2. $gen(id, sid)$ is an operator that takes a pair $(id, sid) : id \in ID, sid \in SID$ and returns a new $id\prime \in ID$.

3. $\sigma : ID \rightharpoonup V$ is a partial function that maps IDs to Values and is used to denote the state of the app. This function keeps track of the values assigned to widget and variable identifiers in a screen.

4. $init : APPID \to SID$ is a function that maps an $appId$ to the start screen identifier sid of an app. The start screen of an app is the screen that the user sees when the app is started for the first time. This function returns the screen ID in an app that needs to be displayed when the app is started.

5. $eval : F \to X$, where $X = V \cup BV$ is a function that returns a *non-boolean value* or a *boolean value* for an operation $f \in F$. This function interprets an operation used in the storyboard.

6. $evalUriVar : ID \rightharpoonup V$ is a partial function that returns the value associated with a variable identifier associated with the URI of a screen. The value is provided by an external app that uses the URI to trigger the corresponding screen. If no external app provides a value for a variable identifier id, then $evalUriVar(id) =\perp$.

7. $outTr : SID \to TR$ is a function that returns all outgoing transitions $tr \in TR$ from a screen $sid \in SID$.

8. $occur : UA \to BV$ is a predicate that is *true* if the user performed a gesture (e.g., button click) and is *false* if the gesture was not performed. This predicate is used to capture any gestures a user may perform on the widgets in a screen visible to the user.

9. $validScr : (SID) \to BV$ is a predicate that is *true* if a given $sid \in SID$ is the ID of a screen in the app; and *false* otherwise.

10. $compr : (h, h') \to BV$ is a predicate that is *true* if $\forall x \in dom(h) : (h(x) = v \wedge h'(x) =\perp) \vee (h(x) =\perp \wedge h'(x) = v)$; *false* otherwise.

11. $stop : (APPID) \to BV$ is a predicate that is *true* if an app with $appId \in APPID$ is moved to the background or is killed explicitly by the user or Android; and *false* otherwise.

12. $\rho : (RN, CN) \to BV$ is a predicate that is *true* if a capability identifier $cn \in CN$ offered by a resource identifier $rn \in RN$ is defined; and *false* otherwise.

13. $erase(h, h')$ is a binary operator that takes two partial functions h, h' and returns another partial function g such that $\forall x \in dom(h') : g(x) = h'(x)$, but $g(x) =\perp$ when $h(x) \neq\perp \wedge h'(x) \neq\perp$. The operator is required to modify the state of the app, σ, when the transition to a screen is taken.

14. $erase_K(h, h\prime)$ is a binary operator that takes two partial functions h, h' and returns another partial function g such that $\forall x \in K, K \subseteq dom(h') : g(x) = h'(x)$, but $g(x) =\perp$ when $h(x) \neq\perp \wedge h'(x) \neq\perp$. The operator is used to change the state of the app, σ, when a transition from a screen to itself is taken.

In addition to the meta-functions and meta-operators, we assume that the custom resources defined in a storyboard are parsed beforehand and stored as the app's custom resources. Hence, resource definitions are not explicitly described in the semantics presented below.

Every syntactic construct defined in Section A is evaluated in a configuration of the form (sid, σ), where sid denotes the *current screen* visible to the user of the app and σ denotes the state of the app when the user is at screen sid. The *initial configuration* of the app $(\perp, \sigma)$.

$$\frac{init(appId) = sid}{\langle appId \ s \ \ r, (\bot, \sigma) \rangle \rightarrow \langle appId \ s \ \ r, (sid, \sigma) \rangle} \tag{1}$$

$$\frac{stop(appId) = true}{\langle appId \ s \ \ r, (sid, \sigma) \rangle \rightarrow (\bot, erase(\sigma, \sigma))} \tag{2}$$

$$\frac{stop(appId) = false \qquad \langle s, (sid, \sigma) \rangle \rightarrow (sid', \sigma')}{\langle appId \ s \ \ r, (sid, \sigma) \rangle \rightarrow \langle appId \ s \ \ r, (sid', \sigma') \rangle} \tag{3}$$

Table 9: App-related Semantic Rules.

App-related rules   The rules listed in Table 9 describe an app's behavior when a user starts an app, interacts with the app, or closes the app.

When an app is started, as per rule 1, it moves from its initial configuration to a configuration where the current screen is set to $sid \in SID$. sid is obtained from the meta-function $init$.

As per rule 2, if an app is stopped by the user or Android, then the terminal configuration $(\bot, \sigma)$, such that $\forall x \in dom(\sigma) : \sigma(x) = \bot$, is reached. This configuration indicates that no screen of the app is visible to the user and the state of the app is undefined.

As per rule 3, an app in a configuration (sid, σ) moves to a new configuration if a screen sid in the app changes the configuration.

Screen-related rules.   The rules listed in Table 10 apply to an app's screens. They are informally described in Section 3.1 as screen extensions.

Rule 4 is applicable when the app is in a configuration (sid, σ) and sid is the identifier of the first screen s_0 in the sequence $s_0 s_1$.

Rule 5 is applicable when the app is in a configuration (sid, σ) and sid is the identifier of some screen in the sequence $s_0 s_1$.

As per rule 6, an app moves from current screen sid with state σ to a screen $sid\prime$ with state $\sigma\prime\prime$ when the widgets in the screen sid extend the state σ to $\sigma\prime$ and one of the outgoing transitions sets the current screen to $sid\prime$ with new state $\sigma\prime\prime$.

As per rule 7, a screen with no outgoing transitions causes the current state σ to change to a new state $\sigma\prime$ when the widgets in the screen sid extend the state σ to $\sigma\prime$.

A proxy screen does not result in any configuration change as per rule 8.

Widget-related rules   The rules listed in Table 11 correspond to linking widgets to data sources in a screen. These rules are informally described as *Widget Extensions* in Section 3.1.

As per rule 9, a sequence of widgets in a screen sid extends the corresponding state σ to $\sigma\prime\prime$ if each widget in the sequence extends σ.

As per rule 10, a widget wid extends state σ with value v if wid is initialized with a variable vid or an operation f, and vid or f evaluates to v in state σ.

As per rule 11, a widget wid does not change the current configuration if wid is initialized with an invalid variable i.e., the variable is undefined in σ.

As per rule 12, a widget wid extends state σ with value v if wid is initialized with the value v.

Transition-related rules   The rules listed in Table 12 specify an app's behavior in the context of outgoing transitions from a screen. They are informally described as *Extensions to Transition* in Section 3.1.

As per rule 13, if the first transition in an ordered set of transitions is taken, then the remaining transitions are not evaluated.

$$\frac{\langle s_0, (sid, \sigma) \rangle \rightarrow (sid', \sigma')}{\langle s_0 s_1, (sid, \sigma) \rangle \rightarrow (sid', \sigma')} \tag{4}$$

$$\frac{\langle s_1, (sid, \sigma) \rangle \rightarrow (sid', \sigma')}{\langle s_0 s_1, (sid, \sigma) \rangle \rightarrow (sid', \sigma')} \tag{5}$$

$$\frac{\langle w, (sid, \sigma) \rangle \rightarrow (sid, \sigma') \qquad \langle tr, (sid, \sigma') \rangle \rightarrow (sid', \sigma'')}{\langle \mathbf{screen}\ sid\ u\ w\ tr, (sid, \sigma) \rangle \rightarrow (sid', \sigma'')} \tag{6}$$

$$\frac{\langle w, (sid, \sigma) \rangle \rightarrow (sid, \sigma')}{\langle \mathbf{screen}\ sid\ u\ w\ , (sid, \sigma) \rangle \rightarrow (sid, \sigma')} \tag{7}$$

$$\langle \mathbf{proxy}\ sid\ appId\ u, (sid, \sigma) \rangle \rightarrow (sid, \sigma) \tag{8}$$

Table 10: Screen-related Semantic Rules.

$$\frac{\langle w_0, (sid, \sigma) \rangle \rightarrow (sid, \sigma') \qquad \langle w_1, (sid, \sigma') \rangle \rightarrow (sid, \sigma'')}{\langle w_0 w_1, (sid, \sigma) \rangle \rightarrow (sid, \sigma'')} \tag{9}$$

$$\frac{\langle x, (sid, \sigma) \rangle \rightarrow v}{\langle wt\ wid\ x, (sid, \sigma) \rangle \rightarrow (sid, \sigma[gen(wid, sid) \mapsto v])}, x ::= vid\ or\ f \tag{10}$$

$$\frac{\langle vid, (sid, \sigma) \rangle \rightarrow \bot}{\langle wt\ wid\ vid, (sid, \sigma) \rangle \rightarrow (sid, \sigma)} \tag{11}$$

$$\langle wt\ wid\ v, (sid, \sigma) \rangle \rightarrow (sid, \sigma[gen(wid, sid) \mapsto v]) \tag{12}$$

Table 11: Widget-related Semantic Rules.

$$\frac{\langle tr_0, (sid, \sigma) \rangle \to (sid', \sigma') \qquad compr(\sigma, \sigma')}{\langle tr_0 tr_1, (sid, \sigma) \rangle \to (sid', \sigma')} \tag{13}$$

$$\frac{\langle tr_0, (sid, \sigma) \rangle \to (sid, \sigma) \qquad \langle tr_1, (sid, \sigma) \rangle \to (sid', \sigma')}{\langle tr_0 tr_1, (sid, \sigma) \rangle \to (sid', \sigma')} \tag{14}$$

$$\frac{\langle ua, (sid, \sigma) \rangle \to true}{\langle b, (sid, \sigma) \rangle \to true \quad validScr(sid') = true \quad \langle p, (sid, \sigma) \rangle \to (sid', \sigma') \quad sid \neq sid'}{\langle \textbf{transition } tid \textbf{ dest } sid' \textbf{ (cond } ua \textbf{ and } b) \ p, (sid, \sigma) \rangle \to (sid', erase(\sigma, \sigma'))} \tag{15}$$

$$\frac{\langle ua, (sid, \sigma) \rangle \to true \quad \langle b, (sid, \sigma) \rangle \to true \quad validScr(sid') = true}{\langle \textbf{transition } tid \textbf{ dest } sid' \textbf{ (cond } ua \textbf{ and } b) \ , (sid, \sigma) \rangle \to (sid', erase(\sigma, \sigma'))} \tag{16}$$

$$\frac{\langle ua, (sid, \sigma) \rangle \to true \quad \langle b, (sid, \sigma) \rangle \to true \quad validScr(sid') = true \quad \langle p, (sid, \sigma) \rangle \to (sid', \sigma')}{\langle \textbf{transition } tid \textbf{ dest } sid \textbf{ (cond } ua \textbf{ and } b) \ p, (sid, \sigma) \rangle \to (sid, erase_{WID}(\sigma, \sigma'))} \tag{17}$$

$$\frac{\langle ua, (sid, \sigma) \rangle \to false}{\langle \textbf{transition } tid \textbf{ dest } sid' \textbf{ (cond } ua \textbf{ and } b) \ p, (sid, \sigma) \rangle \to (sid, \sigma)} \tag{18}$$

$$\frac{\langle b, (sid, \sigma) \rangle \to false}{\langle \textbf{transition } tid \textbf{ dest } sid' \textbf{ (cond } ua \textbf{ and } b) \ p, (sid, \sigma) \rangle \to (sid, \sigma)} \tag{19}$$

Table 12: Transition-related Semantic Rules.

$$\frac{\rho(rn, cn) = true \qquad eval(f) = v}{\langle \mathbf{fun}\ fn\ rn.cn\ , (sid, \sigma) \rangle \rightarrow v} \tag{20}$$

$$\frac{\rho(rn, cn) = true \qquad eval(f) = v}{\langle \mathbf{fun}\ fn\ rn.cn\ a, (sid, \sigma) \rangle \rightarrow v} \tag{21}$$

$$\frac{eval(f) = v}{\langle \mathbf{fun}\ fn\ \ a, (sid, \sigma) \rangle \rightarrow v} \tag{22}$$

$$\frac{eval(f) = v}{\langle \mathbf{fun}\ fn\ , (sid, \sigma) \rangle \rightarrow v} \tag{23}$$

$$\frac{\rho(rn, cn) = true \qquad eval(f) = bv}{\langle \mathbf{fun}\ fn\ rn.cn\ , (sid, \sigma) \rangle \rightarrow bv} \tag{24}$$

$$\frac{\rho(rn, cn) = true \qquad eval(f) = bv}{\langle \mathbf{fun}\ fn\ rn.cn\ a, (sid, \sigma) \rangle \rightarrow bv} \tag{25}$$

$$\frac{eval(f) = bv}{\langle \mathbf{fun}\ fn\ \ a, (sid, \sigma) \rangle \rightarrow bv} \tag{26}$$

$$\frac{eval(f) = bv}{\langle \mathbf{fun}\ fn\ , (sid, \sigma) \rangle \rightarrow bv} \tag{27}$$

Table 13: Operation-related Semantic Rules.

As per rule 14, if the first transition in an ordered set of transitions is not taken, then the remaining transitions in the ordered set are evaluated.

As per rule 15, a transition from a screen to a different screen is taken if the user action associated with it evaluates to *true*, the associated boolean condition evaluates to *true*, and the arguments to the destination screen's parameters p extend the current state σ. Rule 16 is a variant of this rule for a transition without associated input parameters. Rule 17 is a variant of this rule for a transition from a screen to itself.

As per rule 18, a transition is not taken if the user action associated with it evaluates to *false*.

As per rule 19, a transition is not taken if the boolean condition associated with it evaluates to *false*.

Operation-related rules The rules listed in Table 13 are used to evaluate an operation used in the storyboard. These rules correspond to the informal description of *Operations* in Section 3.1.

As per rule 20, an operation f evaluates to a value v, if the meta-function *eval* evaluates f to a non-boolean value v and the resource used in the operation is defined. Rule 21 is a variant of this rule for operations with arguments.

As per rule 22, an operation f evaluates to a value v, if the meta-function *eval* evaluates f to a non-boolean value v. Rule 23 is a variant of this rule for operations without resources or arguments.

$$\frac{\sigma(gen(wid, sid)) = v}{\langle wid, (sid, \sigma)\rangle \to v} \tag{28}$$

$$\frac{\langle occurs(wid, g), (sid, \sigma)\rangle \to bv}{\langle wid.g, (sid, \sigma)\rangle \to bv} \tag{29}$$

$$\frac{\sigma(gen(vid, sid)) = v}{\langle vid, (sid, \sigma)\rangle \to v} \tag{30}$$

$$\frac{\sigma(gen(vid, sid)) = \bot \qquad evalUriVar(gen(vid, sid)) = v}{\langle vid, (sid, \sigma)\rangle \to v} \tag{31}$$

$$\frac{\sigma(gen(vid, sid)) = \bot \qquad evalUriVar(gen(vid, sid)) = \bot}{\langle vid, (sid, \sigma)\rangle \to \bot} \tag{32}$$

$$\frac{\sigma(gen(wid, sid)) = \bot}{\langle wid, (sid, \sigma)\rangle \to \bot} \tag{33}$$

Table 14: ID-related Semantic Rules.

As per rule 24, an operation f evaluates to a boolean value bv, if the meta-function *eval* evaluates f to a boolean bv and the resource used in the operation is defined. Rule 25 is a variant of this rule for operation with arguments.

As per rule 26, an operation f evaluates to a boolean value bv, if the meta-function *eval* evaluates f to a boolean bv. Rule 27 is a variant of this rule for operations without resources or arguments.

ID-related rules   The rules listed in Table 14 are used to evaluate a widget or variable identifiers.

As per rule 28, a widget identifier wid evaluates to a value v under state σ if the state σ maps wid to the value v.

As per rule 29, a gesture g on a widget wid evaluates to *true*, if the gesture occurs on the widget; *false* otherwise.

As per rule 30, a variable identifier vid evaluates to a value v under state σ if the state σ maps vid to the value v.

As per rule 31, a variable identifier vid evaluates to a value v, if $\sigma(vid)$ is undefined, but an external app has provided the value v for vid via $evalUriVar$.

As per rule 32, a variable evaluates to $\bot$ if the variable identifier is undefined in σ and no external app has provided a value for the variable.

As per rule 33, a widget evaluates to $\bot$ if the widget identifier is undefined in σ.

Screen's input parameter-related rules   The rules listed Table 15 are used to evaluate the arguments provided to input parameters of a screen as part of transitions to that screen. An informal description of these rules is provided in Section 3.1 as *Extensions to Transitions*.

As per rule 34, a sequence of syntactic constructs that provide arguments to the parameters of a screen sid extend the state σ, if each construct provides an argument to a parameter, i.e., each construct in the sequence extends the state σ.

$$\frac{\langle p_0, (sid, \sigma) \rangle \rightarrow (sid, \sigma') \qquad \langle p_1, (sid, \sigma') \rangle \rightarrow (sid, \sigma'')}{\langle p_0 p_1, (sid, \sigma) \rangle \rightarrow (sid, \sigma'')} \tag{34}$$

$$\langle \textbf{param } vid\ v, (sid, \sigma) \rangle \rightarrow (sid, \sigma[gen(vid, \phi(vid)) \mapsto v]) \tag{35}$$

$$\frac{\langle x, (sid, \sigma) \rangle \rightarrow v}{\langle \textbf{param } vid_0\ x, (sid, \sigma) \rangle \rightarrow (sid, \sigma[gen(vid_0, \phi(vid_0)) \mapsto v])}, x ::= vid_1, wid,\ or\ f \tag{36}$$

$$\frac{\langle x, (sid, \sigma) \rangle \rightarrow \perp}{\langle \textbf{param } vid_0\ x, (sid, \sigma) \rangle \rightarrow (sid, \sigma)}, x ::= vid_1\ or\ wid \tag{37}$$

Table 15: Screen's input parameter-related Semantic Rules.

As per rule 35, a syntactic construct that provides an argument to a screen's parameter extends the state σ in the current configuration with the value v, if the value v is provided as an argument to parameter ID vid.

As per rule 36, a syntactic construct that provides an argument to a screen's parameter extends the state σ in the current configuration with the value of x, if x has the value v in σ and x is either a variable, widget or an operation.

As per rule 37, A syntactic construct that provides an argument to a screen's parameter does not change the current configuration if the variable or widget that provides the argument is undefined in σ.

Boolean-related rules The rules listed in Table 16 are related to boolean expressions and are similar to traditional notions of conjunction, disjunction, and negation in logical statements.

B.1 Example

We will apply the appropriate semantic rules described above to interpret a portion of the app's storyboard shown in Figure 3. Let us assume that the transition from the screen `Messenger` to the screen `Contacts` is taken. Consequently, via rule 16 the existing configuration changes to (sid, σ), where sid is the identifier of the `Contacts` screen.[4] Since sid in the configuration (sid, σ) is the identifier of `Contacts`, rule 6 is triggered. Consequently, σ is extended with the widgets in the `Contacts` screen via the widget-related rules. Hence, the configuration is now (sid, σ'). At this point, either one of the two outgoing transitions from screen `Contacts` can be taken via rule 13 or 14. Let us assume that the transition from `Contacts` to `SaveStatus` is taken. In this scenario, rule 15 will be triggered, which changes the configuration to (sid'', σ''). In this new configuration sid'' is the identifier of the screen `SaveStatus` and σ'' contains the value of screen `SaveStatus`'s input parameter x. Thereafter, rule 7 is triggered and σ'' is extended with the widgets in `SaveStatus` via the widget-related rules. Since there are no outgoing transitions from `SaveStatus`, the app stays in `SaveStatus` via rule 3 or is terminated via rule 2.

B.2 Safety and Progress

We say that an app is safe if the app does not reach an *invalid configuration* or a *terminal configuration*. Since there are no invalid configurations, an app does not reach an *invalid configuration*. As far as terminal configuration is concerned, we define it as the $(\perp, \sigma)$, such that $\forall x \in dom(\sigma) : \sigma(x) = \perp$. The only time

---

[4]Notice that the transition from `Messenger` to `Contacts` does not have a boolean expression associated with it. For brevity, we assume that the boolean expression b in rule 16 evaluates to true.

$$\frac{\langle b_0, (sid, \sigma) \rangle \rightarrow true \qquad \langle b_1, (sid, \sigma) \rangle \rightarrow true}{\langle b_0 \textbf{ and } b_1, (sid, \sigma) \rangle \rightarrow true} \tag{38}$$

$$\frac{\langle b_0, (sid, \sigma) \rangle \rightarrow false}{\langle b_0 \textbf{ and } b_1, (sid, \sigma) \rangle \rightarrow false} \tag{39}$$

$$\frac{\langle b_1, (sid, \sigma) \rangle \rightarrow false}{\langle b_0 \textbf{ and } b_1, (sid, \sigma) \rangle \rightarrow false} \tag{40}$$

$$\frac{\langle b_0, (sid, \sigma) \rangle \rightarrow true}{\langle b_0 \textbf{ or } b_1, (sid, \sigma) \rangle \rightarrow true} \tag{41}$$

$$\frac{\langle b_1, (sid, \sigma) \rangle \rightarrow true}{\langle b_0 \textbf{ or } b_1, (sid, \sigma) \rangle \rightarrow true} \tag{42}$$

$$\frac{\langle b_0, (sid, \sigma) \rangle \rightarrow false \qquad \langle b_1, (sid, \sigma) \rangle \rightarrow false}{\langle b_0 \textbf{ or } b_1, (sid, \sigma) \rangle \rightarrow false} \tag{43}$$

$$\frac{\langle b, (sid, \sigma) \rangle \rightarrow false}{\langle \textbf{ not } b, (sid, \sigma) \rangle \rightarrow true} \tag{44}$$

$$\frac{\langle b, (sid, \sigma) \rangle \rightarrow true}{\langle \textbf{ not } b, (sid, \sigma) \rangle \rightarrow false} \tag{45}$$

Table 16: Boolean-related Semantic Rules.

an app reaches the terminal configuration is when the app is stopped by the user or Android. In this configuration no other transition is possible and the app is said to be terminated. *Since no rule defined above, except rule 2, makes an app reach the terminal configuration, we say that the semantics ensure safety by construction.*

The semantic rules defined above ensure that an app is never "stuck", i.e., if an app is in a valid configuration or a non-terminal configuration, then the app will either terminate or will be evaluated further in some configuration. We call this property of not getting stuck as progress. Formally, progress is specified as the following theorem:

Theorem B.1. *If an app is in a configuration (sid, σ), then $\exists (sid', \sigma') : \langle app, (sid, \sigma) \rangle \rightarrow \langle app, (sid', \sigma') \rangle$, or $\langle app, (sid, \sigma) \rangle \rightarrow (\bot, erase(\sigma, \sigma))$*

We need the following lemma to prove theorem B.1.

Lemma B.2. *If an app is in a configuration (sid, σ) and $\forall tr \in TR : outTr(sid) = tr$, then $\exists (sid', \sigma\prime) : \langle tr, (sid, \sigma) \rangle \rightarrow (sid', \sigma')$, where $(sid', \sigma\prime)$ may or may not equal (sid, σ).*

Proof. We prove Lemma B.2 by induction on the structure of tr.

Base Case: When tr is one transition from the screen sid, Lemma B.2 is trivially true since either Rule 15, 16, 17, 18, or 19 will always apply.

Inductive Case: When tr is an ordered set of transitions, $tr_0 tr_1$, from the screen sid, then either apply rule 13 and 14 with the induction hypothesis on tr_0 or apply 14 with the induction hypothesis on tr_1.

Hence, by the principle of induction, Lemma B.2 is true.

□

Proof. Let us assume that an app is in (sid, σ), where sid is a screen in the app and σ is the state of the app at screen sid.

Assume that there is an ordered set of transitions tr from screen sid, then by lemma B.2, $\exists (sid', \sigma')$ such that the app will be further evaluated in (sid', σ') as per rule 3. If the app is stopped, then the app will terminate as per rule 2.

□

C   Information Flow Analysis Algorithm

This analysis tracks the flow of information in the form of values from *sources* to *sinks* in a storyboard, as defined in Section 3.3. In this section, We formally describe the algorithm.

Meta-language: Before specifying the algorithm, we define the following functions and structures required to understand it and the corresponding proof of correctness.

1. $Idn : Exp \rightarrow IDN$, where $Exp = W \cup P \cup F$ and $IDN = WID \cup VID \cup FN$, is a function is used to obtain the identifier of a widget, input parameter, or an operation. It is defined as follows:

   (a) *Widget* : if $w \in Exp$ and $w ::= wt\ wid\ x$, then $Idn(w) = gen(wid, \phi(wid))$

   (b) *Parameter* : if $p \in Exp$ and $p ::= $ **param** $vid\ x$, then $Idn(p) = gen(vid, \phi(vid))$

   (c) *Operation* : if $f \in Exp$ and $f ::= $ **fun** $fn\ rn.cn\ a$, then $Idn(f) = fn$

2. $FV : Exp \rightarrow 2^{IDN}$, is a function that maps $e \in Exp$ to the power set of IDN. FV is used to retrieve the identifiers that are used as values of a widget, input parameter, or operation. It is defined as follows:

   (a) if $e \in F$, then

       i. if $e ::= $ **fun** $fn\ rn.cn\ vid$, then $FV(e) = \{gen(vid, \phi(vid))\}$

       ii. if $e ::= $ **fun** $fn\ rn.cn\ wid$, then $FV(e) = \{gen(wid, \phi(wid))\}$

       iii. if $e ::= $ **fun** $fn\ rn.cn\ f\prime$, then $FV(e) = \{Idn(f\prime)\}$

  iv. if $e ::= \textbf{fun } \textit{fn rn.cn } a_0 a_1$, then $FV(e) = FV(f_0) \cup FV(f_0)$,
   where $f_0 ::= \textbf{fun } \textit{fn rn.cn } a_0$ and $f_1 ::= \textbf{fun } \textit{fn rn.cn } a_1$
  v. otherwise, $FV(e) = \emptyset$

 (b) if $e \in W$, then

  i. if $e ::= \textit{wt wid vid}$, then $FV(e) = \{gen(vid, \phi(vid))\}$
  ii. if $e ::= \textit{wt wid f}$, then $FV(e) = \{Idn(f)\}$
  iii. otherwise, $FV(e) = \emptyset$

 (c) if $e \in P$, then

  i. if $e ::= \textbf{param } vid_0 \, vid_1$, then $FV(e) = \{gen(vid_1, \phi(vid_1))\}$
  ii. if $e ::= \textbf{param } vid \, wid$, then $FV(e) = \{gen(wid, \phi(wid))\}$
  iii. if $e ::= \textbf{param } vid \, f$, then $FV(e) = \{Idn(f)\}$
  iv. otherwise, $FV(e) = \emptyset$

3. The *safe* relation is used to collect direct flows between sources and sinks in the storyboard that have been marked by a developer with the *safe* keyword. Section A shows the syntax for marking direct flows as *safe*.

4. *safeParams* denotes the set of input parameters of a proxy screen $ps \in PS$ such that ps has been marked *safe* or ps has the *app* attribute set.

5. $CV : F \rightarrow BV$ is a predicate such that $CV(f) = true$ if operation f is an untrusted sink; $CV(f) = false$ otherwise.

6. $IV : F \rightarrow BV$ is a predicate such that $IV(f) = true$ if operation f is an untrusted source; $IV(f) = false$ otherwise.

Algorithm: The analysis algorithm proceeds in two stages. First, it captures the *direct* relationships between the identifier of a widget, an input parameter, or an operation with the identifier used as value in a widget, input parameter, or an argument to an operation in a binary relation called *influences*. It then uses *influences* to build a reflexive transitive closure $influences^*$ to capture the *indirect* relationships between widgets, input parameters, and operations. Second, for each widget, operation, and input parameter expression, the analysis collects their identifiers if they are directly or indirectly influenced by unsafe and untrusted identifiers. A violation is detected if a widget, input parameter, or operation has a non-empty set of such identifiers associated with it.

 The algorithm is formally defined in the following steps:

1. Calculate the binary relation *influences*: $IDN \rightarrow IDN$ for each $e \in W \cup P \cup F$ as follows:

 (a) *Widgets $w \in W$*:

  i. if $w ::= \textit{wt wid vid}$, then $\{(gen(vid, \phi(vid)), gen(wid, \phi(wid)))\}$.
  ii. if $w ::= \textit{wt wid f}$, then $\{(Idn(f), gen(wid, \phi(wid)))\}$.
  iii. if $w ::= w_0 w_1$, then $influences_{w0} \cup influences_{w1} \cup influences$

 (b) *screen input parameters $p \in P$*:

  i. if $p ::= \textbf{param } vid_0 \, vid_1$, then $\{(gen(vid_1, \phi(vid_1)), gen(vid_0, \phi(vid_0)))\}$
  ii. if $p ::= \textbf{param } vid \, wid$, then $\{(gen(wid, \phi(wid)), gen(vid, \phi(vid)))\}$
  iii. if $p ::= \textbf{param } vid \, f$, then $\{(Idn(f), gen(vid, \phi(vid)))\}$
  iv. if $p ::= p_0 p_1$, then $influences_{p0} \cup influences_{p1} \cup influences$

 (c) *Operations $f \in F$*:

  i. if $f ::= \textbf{fun } \textit{fn rn.cn } vid$, then $\{(gen(vid, \phi(vid)), fn)\}$
  ii. if $f ::= \textbf{fun } \textit{fn rn.cn } wid$, then $\{(gen(wid, \phi(wid)), fn)\}$.
  iii. if $f ::= \textbf{fun } \textit{fn rn.cn } f\prime$, then $\{(Idn(f\prime), fn)\}$

iv. if $f ::= \textbf{fun } fn\, rn.cn\, a_0 a_1$, then $influences_{f0} \cup influences_{f1} \cup influences$,
where $f_0 ::= \textbf{fun } fn\, rn.cn\, a_0$ and $f_1 ::= \textbf{fun } fn\, rn.cn\, a_1$

2. Calculate $influences^*$, the reflexive transitive closure of the binary relation $influences$.

3. Mark untrusted sources using the $Untrusted : IDN \to BV$ predicate for each $e \in F \cup U$ as follows:

   (a) if $e \in F \wedge IV(e)$, then $\forall(Idn(f), k) \in influences^* : Untrusted(k) = true$

   (b) if $e ::= us/K$, then $\forall k \in K : Untrusted(k) = true \wedge \forall(gen(vid, \phi(vid)), y) \in influences^* : Untrusted(y) = true$

4. Collect flows marked $safe$ for each $e \in W \cup F \cup P$ as follows:

   (a) if $e ::= safe\, wt\, wid\, x$, then $\{x, gen(wid, \phi(wid)))\} \cup safe$ and $\forall(gen(wid, \phi(wid)), k) \in influences^* : \{(gen(wid, \phi(wid)), k)\} \cup safe$

   (b) if $e ::= w_0 w_1$, then $safe_{w0} \cup safe_{w1} \cup safe$

   (c) if $e ::= \textbf{fun } fn\, rn.cn\, safe\, x$, then $\{(gen(x, \phi(x)), fn)\} \cup safe$, where $x \in (WID \cup VID)$

   (d) if $e ::= \textbf{fun } fn\, rn.cn\, safe\, f$, then $\{(Idn(f), fn)\} \cup safe$

   (e) if $e ::= \textbf{fun } fn\, rn.cn\, a_0 a_1$, then $safe_{e0} \cup safe_{e1} \cup safe$, where $e0 ::= \textbf{fun } fn\, rn.cn\, a_0$ and $e1 ::= \textbf{fun } fn\, rn.cn\, a_1$

   (f) if $e ::= \textbf{param } vid\, x$ and $gen(vid, \phi(vid)) \in safeParams$, then $\{(x, gen(vid, \phi(vid))) \in safe\}$

   (g) if $e ::= p_0 p_1$, then $safe_{p0} \cup safe_{p1} \cup safe$

5. Calculate the function $IF : Exp \to 2^{IDN}$ for each $e \in W \cup F \cup P$ as follows:

   (a) if $e \in W$, then

      i. if $e ::= wt\, wid\, vid$, then
      $IF(e) = \{gen(wid, \phi(wid)) \mid Untrusted(vid) \wedge (((vid, wid) \notin safe) \wedge (\exists(z, vid) : (z, vid) \in influences \implies (\forall(y, z) : (y, z) \in influences^* \implies (y, z) \notin safe)))$

      ii. if $e ::= wt\, wid\, f$, then
      $IF(e) = \{gen(wid, \phi(wid)) \mid IV(f) \wedge (Idn(f), wid) \notin safe$

      iii. if $e ::= w_0 w_1$, then $IF(e) = IF(w_0) \cup IF(w_1)$

   (b) if $e \in F$, then

      i. if $e ::= \textbf{fun } fn\, rn.cn\, wid$, then
      $IF(e) = \{fn \mid (wid, fn) \notin safe \wedge Idn(w) = wid \wedge IF(w) \neq \emptyset\}$
      $\cup \{fn \mid CV(e) \wedge ((wid, fn) \notin safe) \wedge (\exists(fn', wid) : (fn', wid) \in influences \implies (fn', wid) \notin safe) \wedge (\exists(y, vid), (vid, wid) : ((y, vid) \in influences \wedge (vid, wid) \in influences) \implies (\forall(k, vid) : (k, vid) \in influences^* \implies (k, vid) \notin safe))\}$

      ii. if $e ::= \textbf{fun } fn\, rn.cn\, vid$, then
      $IF(e) = \{fn \mid (Untrusted(vid) \vee CV(e)) \wedge (vid, fn) \notin safe \wedge (\exists(z, vid) : (z, vid) \in influences \implies (\forall(y, z) : (y, z) \in influences^* \implies (y, z) \notin safe))\}$

      iii. if $e ::= \textbf{fun } fn\, rn.cn\, f$, then
      $IF(e) = \{fn \mid (IV(f) \vee CV(e)) \wedge (Idn(f), fn) \notin safe\}$

      iv. if $e ::= \textbf{fun } fn\, rn.cn\, a_0 a_1$, then
      $IF(e) = IF(e_0) \cup IF(e_1)$, where $e_0 ::= \textbf{fun } fn\, rn.cn\, a_0$ and $e_1 ::= \textbf{fun } fn\, rn.cn\, a_1$

   (c) if $e \in P$, then

      i. if $e ::= \textbf{param } vid\, id$, where $id \in (WID \cup VID)$, then
      $IF(e) = \{gen(vid, \phi(vid)) \mid (id, vid) \in safe \vee (\forall(y, id) \in influences^* : (y, id) \notin safe)\}$

      ii. if $e ::= \textbf{param } vid\, f$, then $IF(e) = \{gen(vid, \phi(vid)) \mid Idn(f) = fn \wedge (fn, vid) \in safe$

      iii. if $e ::= e_0 e_1$, then $IF(e) = IF(e_0) \cup IF(e_1)$

Exp	IF	IF (safe)
TextView Phone y	{gen(Phone,$\phi(Phone)$))}	{}
TextView Status "MsgSent"	{}	{}
TextView Status dispMsg(x)	{}	{}
sendMsg(...)	{}	{}
getContacts(...)	{}	{}
dispMsg(x)	{Idn(dispMsg)}	{}
param z Phone	{}	{}
param x Phone	{}	{}

Table 17: Information Flow Analysis of Storyboard shown in Figure 3. The first column indicates an $e \in Exp$ (i.e., widget, operation, or input parameter of a screen). The second column indicates a set obtained by evaluating IF(e) assuming that nothing was marked as *safe*. The third column indicates IF with a flow marked as *safe*.

The analysis reports a violation if $\exists e \in Exp : IF(e) \neq \emptyset$ in the storyboard.

As an example, consider information flow analysis of the storyboard in Figure 3. As per the algorithm, we first build the binary relation:

$$influences = \{(y, Phone), (Phone, z), (Phone, x), (dispMsg, Status), (x, dispMsg),$$
$$(Phone, savePhone), (getContacts, sendMsg)\}$$

Next, a reflexive transitive closure is calculated as follows:

$$influences^* = \{(y, Phone), (Phone, z), (Phone, x), (dispMsg, Status), (x, dispMsg),$$
$$(Phone, savePhone), getContacts, sendMsg), (y, z), (y, x), (Phone, dispMsg),$$
$$(y, dispMsg), (y, Status), (x, Status)), ...\}$$

The predicates IV and CV return *false* for every operation in the storyboard since none of them use resources that correspond to untrusted source/sink. Since, y is a variable from an external app, $\forall(y, k) \in influences^* : Untrusted(y, k) = true$.

Table 17 shows the result of computing IF for each $e \in Exp$ in the storyboard before any flow was marked as *safe*. Since $\exists e \in Exp : IF(e) \neq \emptyset$, the analysis reports a violation. To fix the violation, the variable y needs to be obtained from a trusted source or the flow/s related to y should be marked *safe*. If we take the latter approach and mark the flow between y and the widget *Phone* as *safe* as shown in Figure 3, then on running the analysis again, *safe* is as follows:

$$safe = \{(y, Phone), (Phone, z), (Phone, x), ...\}$$

Since $\forall e \in Exp : IF(e) = \emptyset$ as shown in Table 17 third column, no violations are reported by the analysis.

C.1 Proof of Correctness

The correctness of the function IF hinges on the calculation of the transitive closure $influences^*$, predicate *Untrusted*, and set *safe*. Since *Untrusted* and *safe* are based on a pre-defined list of trusted and untrusted identifiers and developer-provided annotations/indicators, it is enough to prove that the relations captured in $influences^*$ reflect the flow as specified in the semantics.

Theorem C.1. $\forall a \in APP, \forall sid, sid' \in SID, x, y \in ID, \sigma, \sigma' : \langle(a), (sid, \sigma)\rangle \rightarrow^* \langle a, (sid', \sigma')\rangle \wedge \sigma(x) = \sigma'(y) \implies (x, y) \in influences^*$

Proof. We prove this by induction on the no. of steps it takes an app from the current configuration (sid, σ) to reach a configuration (sid', σ').

Base Case: Assume that a is an app, $sid, sid' \in SID, x, y \in ID, \sigma, \sigma'$ such that $\langle a, (sid, \sigma) \rangle \rightarrow \langle a, (sid', \sigma') \rangle \land \sigma(x) = \sigma'(y)$. In screen sid', $\exists l$ such that l is a widget or provides an argument to an input parameter of another screen as part of a transition from sid'. Since $\sigma(x) = \sigma'(y)$, $y = Idn(l)$ and $k \in FV(l)$, where either $k = x$ or k is the identifier of an operation with an argument x. By the definition of *influences*, either $(x, y) \in influences$ or $(x, k) and (k, y) \in influences$. In either case, $(x, y) \in influences^*$.

Therefore, C.1 is true.

Inductive Case:

- Case 1: Let's assume that a is an app, $sid, sid' \in SID, x, z \in ID, \sigma, \sigma''$ such that $\langle (a), (sid, \sigma) \rangle \rightarrow^k \langle a, (sid', \sigma'') \rangle \land \sigma(x) = \sigma''(z) \implies (x, z) \in influences^*$, where k is the no. of steps it takes to reach from σ to σ''. This is our induction hypothesis.

  Further, assume that $\exists sid'' \in SID, y \in ID, \sigma'$ such that $\langle a, (sid', \sigma'') \rangle \rightarrow \langle a, (sid'', \sigma') \land \sigma''(z) = \sigma'(y)$ and l is a widget in screen sid'' or provides an argument to the input parameter of a screen via a transition from sid''. Since $\sigma''(z) = \sigma'(y)$, $y = Idn(l)$ and $k \in FV(l)$, where $k = z$ or k is the identifier of an operation that uses z as an argument. In either case, $(z, y) \in influences$ due to the definition of *influences*. Since $influences \subseteq influence^*$, $(z, y) \in influences^*$. From the induction hypothesis, $(x, z) \in influences^*$. Since, $influences^*$ is a transitive closure, $(x, y) \in influences^*$.

- Case 2: Let's assume that a is an app, $sid, sid' \in SID, \sigma, \sigma'$ such that $\langle (a), (sid, \sigma) \rangle \rightarrow^m \langle a, (sid'), \sigma' \rangle \land \sigma(x) = \sigma'(z) \implies (x, z) \in influences^*$, and $sid'' \in SID, \sigma''$ such that $\langle (a), (sid', \sigma') \rangle \rightarrow^n \langle a, (sid'', \sigma'') \rangle \land \sigma'(z) = \sigma''(j) \implies (z, j) \in influences^*$, and $k = m + n$.

  Further, assume that $\exists sid''' \in SID, y \in ID, \sigma''' : \langle a, (sid'', \sigma'') \rangle \rightarrow \langle a, (sid''', \sigma''') \land \sigma''(j) = \sigma'''(y)$ and l is a widget in screen sid''' or provides the argument to an input parameter of a screen via a transition from sid'''. Since $\sigma''(j) = \sigma'''(y)$, $y = Idn(l)$ and $k \in FV(l)$, where $k = j$ or k is the identifier of an operation that uses j as an argument. In either case, $(j, y) \in influences$ by the definition of *influences*. Since $influences \subseteq influence^*$, $(j, y) \in influences^*$. From the induction hypothesis, $(x, z) \in influences^*$ and $(z, j) \in influences^*$. Since, $influences^*$ is a transitive closure, $(x, y) \in influences^*$.

Conclusion: By the principle of induction C.1 is true.

□

The converse of Theorem C.1 does not hold for the analysis because the analysis does not consider the effects of constraints when building the *influences* relation. So, it is possible that an ID x *influences* y even if the semantics does not allow x to flow into y. For example, let us assume that a transition t is guarded by a constraint b and when b is *true*, an argument a is passed to the destination screen. Let's also assume that $x \in FV(a)$ and $y = Idn(a)$. Finally, let us assume that b is always *false*. In such a scenario, x *influences* y but the semantics will not allow x to flow into y. This implies that the analysis will flag violations even if there isn't any. However, the developer can override the violation by setting the *safe* attribute appropriately.