How useful are existing monitoring languages for securing Android apps?

Steven Arzt[1], Kevin Falzon[1], Andreas Follner[1], Siegfried Rasthofer[1], Eric Bodden[1] and Volker Stolz[2]

[1] Secure Software Engineering Group, EC SPRIDE, Technische Universität Darmstadt, Germany
[2] Precise Modelling and Analysis Group, University of Oslo, Norway

Abstract: The Android operating system is currently dominating the mobile device market in terms of penetration and growth rate. An important contributor to its success are a wealth of cheap and easy-to-install mobile applications, known as *apps*. Today, installing *untrusted* apps is the norm, though this comes with risks: malware is ubiquitous and can easily leak confidential and sensitive data.

In this work, we investigate the extent to which we can specify complex information flow properties using existing specification languages for runtime monitoring, with the goal to encapsulate potentially harmful apps and prevent private data from leaking. By modelling a set of representative, Android-specific security policies with Tracematches, JavaMOP, Dataflow Pointcuts and PQL, we are able to identify policy-language features that are crucial for effectively defining runtime-enforceable Android security properties.

Our evaluation demonstrates that while certain property languages suit our purposes better than others, they all lack essential features that would, if present, allow users to provide effective security guarantees about apps. We discuss those shortcomings and propose several possible mechanisms to overcome them.

1 Introduction

According to a recent study [Cor12], Android now has about 75% market share in the mobile-phone market, with a 91.5% growth rate over the past year. With Android phones being ubiquitous, they become a worthwhile target for security and privacy violations. Attacks range from broad data collection for the purpose of targeted advertisement, to targeted attacks, such as the case of industrial espionage. Attacks are most likely to be motivated primarily by a social element: a significant number of mobile-phone owners uses their device both for private and work-related communication [Bit12]. Furthermore, the vast majority of users install apps containing code whose trustworthiness they cannot judge and which they cannot effectively control.

These problems are well known, and indeed the Android platform does implement state-of-the-practice measures to impede attacks. The Android platform is built as a stack, with various layers running on top of each other [And12].

The lower levels consist of an embedded Linux system and its libraries, with Android applications residing at the very top. Users typically acquire these applications through various channels (e.g., the Google Play marketplace[1]). The underlying embedded Linux system provides the enforcement mechanisms common to the Linux kernel, such as a user-based permission model, process isolation and secure inter-process communication. By default, an application is not allowed to directly interact with other applications, operating system processes, or a user's private data [Goo12]. The latter includes, for example, access to the contacts list. Access to such private data is regulated by Android via a *permission-based* security model, where applications have to statically declare the permissions they require in order to access security-sensitive API functions. An application may only be installed following the user's informed consent, yet users currently have little control over the installation process, as they must either grant all of the permissions that an app demands, or else forego installation.

Popular Android extensions such as AppGuard [BGH$^+$12] mitigate this problem by instrumenting apps with dynamic permission checks at installation time. Through this mechanism, users can revoke permissions they had initially granted at installation time. While a definite improvement, permission revocation is not a complete solution. Part of the problem originates from the coarse-grained nature of Android permissions [NKZ10, ZZJF11]. For instance, an app may require internet and phone-book access permissions to function correctly, in which case revoking either would not be an option. Nevertheless, one may wish to forbid the app from transmitting phone book entries over the internet. Such fine-grained restrictions on information flow are not possible with Android's existing permission-based security model.

One possible solution to the problem are specialized *runtime monitoring approaches* for information flow properties. In the past, researchers have proposed several different monitor specification languages for specifying policies that, through a runtime monitoring tool, are automatically enforced as the monitored program executes. These tools typically instrument the program artifact directly rather than its source code. Given a policy definition and a potentially unsafe or insecure program, a specialized compiler or weaver instruments the program with security checks which ensure that every program execution that violates the policy is detected. Developers can use this information, e.g. for logging violations or defining countermeasures to abort or gracefully handle problematic executions.

Yet, how well do these specification languages address the problems at hand? Can any of the currently available languages be used to effectively specify and enforce security and privacy policies that are of practical interest in the context of Android? In this work, we aim to answer this question by evaluating four different monitor specification languages. Our work focuses on languages, rather than tools, because we believe that approaches for information-flow analysis need to be customizable, and that users of those tools should be able to prove security guarantees over monitored apps. Languages have the potential to allow for a level of abstraction that permits such proofs.

We specifically exclude pure information-flow analysis tools that do not offer a policy language, for instance TaintDroid [EGgC$^+$10]. While such tools present important backend

[1]Available at `https://play.google.com/` (December 2012)

technology that allows for efficient analysis, in this paper we aim to answer important questions on specification languages, i.e., on frontend technology. With respect to tooling, TaintDroid also differs from the approaches studied here in that it requires modification to the Android runtime, which in an end-user scenario is undesired. All approaches presented here work by instrumenting the app under test; the instrumented app can then execute in a standard execution environment.

We have formulated different Android code snippets that include information-flow properties and tried to enforce security properties of them using the finite-state monitoring languages Tracematches [AAC$^+$05] and JavaMOP [CR07], the AspectJ language extension Dataflow Pointcuts [MK03, ABB$^+$09] and the program query language PQL [MLL05]. We found that some of the languages are effective at securing well-structured programs against input-driven attacks such as SQL injections, a concern typically reserved for server applications. However, in our scenario we must assume that it is the app, not the input, that is malicious, and hence we cannot expect programs to be well-structured, as they can be arbitrarily obfuscated. None of the existing approaches allow security experts to define policies in a way that would allow detecting violations in such a setting. We discuss this problem in detail and present a range of possible solutions that designers of policy languages can choose from to mitigate the problem.

To summarize, this work presents the following original contributions:

- an investigation of the suitability of Tracematches, JavaMOP, Dataflow Pointcuts and PQL for the monitoring of security policies,
- a discussion of the reasons why these languages fail to provide practically enforceable security guarantees, and
- a discussion of possible language-design options that could mitigate the problem covering the areas of recursion and reuse, variable and member access, custom monitor states, global and persistent state, for implicit information flow detection, handling of primitive data types and native code, placement of sanitizers, “proceed” instructions and specification language typing.

2 Example Properties

Recently, several works [ZJ12, EOMC11, EGgC$^+$10, GCEC12, FHM$^+$12, KB12] have investigated the detection of different forms of vulnerabilities in Android apps. A principal concern is the leakage of sensitive information, including the IMEI, IMSI[2] and location information. In addition, there are malicious applications that harm the user financially, for instance by sending premium-rate SMS/MMS messages [EOMC11], or eavesdropping on online banking transactions, stealing the mTAN[3] and withdrawing money from the victim’s account [KB12]. Furthermore, Fahl et al. [FHM$^+$12] found various forms of SSL/TLS misuse in Android applications.

[2]International Mobile Equipment/Subscriber Identity

[3]One-time password sent to a user’s phone by an online banking service to authorize a transaction.

The most prevalent vulnerabilities in the mobile scenario are related to information-flow properties, such as sending sensitive information to a specific target. Therefore, we created three different kinds of malicious code snippets (pseudocode) that could be implemented in real Android applications. These snippets will be used as a basis for our evaluation to demonstrate the respective strengths and weaknesses of the different property languages. Listing 1 shows an example of an information-flow property. The sensitive information IMEI together with the location is read from the device and is propagated through the code until it arrives at the "sendTextMessage" sink. At this point the sensitive information is leaked because it leaves the device in form of an SMS message to a specific phone number. This leads to an explicit information-flow violation in the present example. However, Listing 2 shows almost the same example (excluding the location information), but with the difference of an implicit flow. The example converts the IMEI into a bit stream and iterates over the bits. There exists an implicit flow in line 8 for the string argument "0" that is only sent to the specific target under the condition that the bit is 0. Line 10 performs the corresponding action in case the bit is 1, sending the string "1".

```
 1  String imei =
         TelephonyManager.getDeviceID();
 2
 3  double latitude =
         LocationManager.getLatitude();
 4
 5  String message = "IMEI: " + imei;
 6  message += "LOCATION: " +
         latitude;
 7
 8  sendTextMessage("+44 020 7321
         0905", message);
 9
10 ...
```

Listing 1: Violation of an information-flow property (explicit flow)

```
 1  String imei =
         TelephonyManager.getDeviceID();
 2
 3  int[] imeiAsBitStream =
         imei.toBitStream();
 4
 5  for(int bit : imeiAsBitStream)
 6    if(bit == 0)
 7      sendTextMessage("+44 020 7321
             0905", "0");
 8    else
 9      sendTextMessage("+44 020 7321
             0905", "1");
10 ...
```

Listing 2: Violation of an information-flow property (implicit flow)

The example in Listing 3 shows a code snippet that iterates through every phone number in the mobile device's contact list and sends a spam message ("I love you!") to these numbers. In contrast with the previous examples, countermeasures for this code will be more concerned with the frequency of transmission of messages, rather than information leaking into messages. For instance, the policy *"allow at most 3 text messages to be sent from a specific app per day"* would require some kind of counting mechanism in the monitoring language.

```
1  String[] contactPhoneNumbers = getAllPhoneNumbersOfContacts();
2
3  for(String number : contactPhoneNumbers)
4    sendTextMessage(number, "I love you!");
5 ...
```

Listing 3: Spam message to all contacts

3 Tracematches

Tracematches [AAC$^+$05] are an extension to the aspect-oriented programming language AspectJ [KHH$^+$01]. The extension provides users with the ability to define runtime monitors that can match the program's dynamic execution trace against a regular expression of events. Tracematches support free variables in those events, which allows users to relate events to one another on a consistent group of objects. This makes Tracematches an ideal specification language for defining properties in cases where each such group of objects, as well as the state that they share, is finite.

Listing 4 details a compact Tracematch designed to detect and prevent the SMS-spamming behaviour described in Listing 3. The property is triggered whenever two consecutive SMS messages are sent with no intervening user interaction. Security events of interest are defined using AspectJ pointcuts, which map points in the program to symbolic event names. In this case, a pointcut for the SMS sending method is specified, along with a pointcut `userInteraction` defined elsewhere.

The sequence of interest is defined as a regular expression, where events are received and matched against the defined expression. The expression `send_sms send_sms` matches a trace "`send_sms send_sms`" but not "`send_sms user_input send_sms`". Therefore, the error handler would trigger exactly if two consecutive SMS messages are sent without an intervening user action. When the handler triggers, it replaces the call to `sendTextMessage`.

Tracematches are not suitable, however, for defining information-flow properties. This is due to the fact that information flow can propagate through an arbitrary number of variables and objects. While it is possible to define a Tracematch that can detect insecure information flows for any particular code example, one often finds that such properties do not scale very well, and can be very fragile. For example, consider Listing 5, which describes a Tracematch designed to detect the violation of the explicit flow property presented in Listing 1. In Java, string concatenation via the $+$ operator is implemented as a series of append operations on a StringBuilder structure. As a result, the Tracematch must attempt to follow the sensitive data elements as they traverse multiple structures. Consequently, even a seemingly innocuous change of concatenation operators would foil detection. Similarly, detection would be avoided were one to rearrange the sequence in which the concatenation operators take place, or by introducing additional intermediate steps, which may also re-encode the sensitive information, making it harder to keep track of propagations. It is also not possible to generalize a Tracematch such that it would track flows of arbitrary length: any given Tracematch can only reason about a fixed number of values, but information flows can involve an arbitrary number of objects.

```
tracematch() {
    sym user_input before: userInteraction();
    sym send_sms   around: call(* SmsSession.sendTextMessage(..));

    send_sms send_sms {
        System.err.println("Sms spam detected! Sending aborted."); }
}
```

Listing 4: Tracematch detecting SMS spam

```
tracematch(String imei, double lat, Object msg1, Object msg2, Object msg2_sb,
     Object merged, String msg1_s, String msg2_s, String merged_s) {
  sym ret_imei   after returning(imei):    call(* TelephonyManager.getDeviceID());
  sym ret_lat    after returning(lat):     call(* LocationManager.getLatitude());
  // Trigger when appending sensitive data to a string
  sym appendI    after returning(msg1):    call(* *.append(..)) && args(imei);
  sym appendL    after returning(msg2):    call(* *.append(..)) && args(lat);
  // Translating from internal representation to strings ("message" is a String)
  sym appendI_s after returning(msg1_s):  call(* *.toString()) && target(msg1);
  sym appendL_s after returning(msg2_s):  call(* *.toString()) && target(msg2);
  sym merge_s    after returning(merged_s):call(* *.toString()) && target(merged);
  // Trigger on conversion from String to internal representation
  sym appendL_sb  after returning(msg2_sb):call(*.new(..)) && args(msg1_s);
  // Sensitive data (IMEI and Latitude) have been merged into a single string
  sym merge_sb   after returning(merged):  call(* *.append(..)) && target(msg2_sb)
       && args(msg2_s);
  // Sensitive data leaked via SMS
  sym sendSms    around: call(* SmsSession.send*(..)) && args(*, merged_s);

  // The regular expression
  ret_imei ret_lat appendI appendI_s+ appendL_sb appendL appendL_s+ merge_sb
       merge_s sendSms {
    proceed(sanitize(merged_s)); //send a sanitized version of the SMS
  }
}
```

Listing 5: Tracematch detecting explicit information flow of secret strings

4 JavaMOP

JavaMOP [CR07] is designed to monitor properties defined in a range of different temporal logics. Listing 6 is a reformulation of the security property for detecting SMS spamming. Similar to the Tracematch property described in Listing 4, this property identifies a set of methods related to user interaction, along with the method for transmitting a message. In this example, the property has been expressed as a *Linear-Temporal Logic* (LTL) formula, which states that a `send_sms` event should be followed by a `user_input`.

Listing 7 describes an approach to performing taint tracking in the context of the problem defined in Listing 1. The property definition is less fragile than the Tracematch definition. In particular, it completely abstracts from the order in which events occur. Also, by using a custom data structure `taintedStrings` with membership queries, the specification can define the property recursively: a string is tainted if it is returned from a source or it is built from a tainted string. As HashSets use equality, and not identities, this solution may flag (string-representations of) values as tainted *regardless of their source*.

While JavaMOP improves on the Tracematch specification, it is interesting to note that

```
SmsSpam() {
  event user_input before() : userInteraction() { }
  event send_sms before() :  call(* SmsSession.sendTextMessage(..))  { }
  // If SMS sent, next event must be an interaction
  ltl: [](send_sms => o user_input)

  @violation { System.err.println("Sms spam detected!"); }
}
```

Listing 6: JavaMOP (LTL plugin) detecting SMS spam

```
ExplicitSpec() {
  Set <String> taintedStrings = new HashSet<String>();

  void taint(String s) { taintedStrings.add(s); }

  boolean isTainted(String s) { return taintedStrings.contains(s); }

  event retImei after() returning (String imei):
    call(* TelephonyManager.getDeviceID())  { taint(imei); }

  event retLat  after() returning (double lat):
    call(* LocationManager.getLatitude())  { taint(new Double(lat).toString()); }

  event propagate_strings after(StringBuilder sb, String s):
    (call(* StringBuilder.append(String)) || call (StringBuilder.new(String)))
        && target(sb) && args(s) {
      if (isTainted(s)) taint(sb.toString());
  }

  event propagate_doubles after(StringBuilder sb, double d):
    (call(* StringBuilder.append(double )) || call (StringBuilder.new(double)))
        && target(sb) && args(d) {
      String s = new Double(d).toString();
      if(isTainted(s)) taint(sb.toString());
    }

  event sink before(String s):
    call (* sendTextMessage(String, String)) && args(t, s) {
      if (isTainted(s))
        System.err.println("Sensitive information (" + s + ") is sent to:" + t);
    }
}
```

Listing 7: Taint tracking using events in JavaMOP

Listing 7 basically uses no JavaMOP features any longer: the same monitor could just as well have been written in plain AspectJ.

5 Dataflow Pointcuts

Alhadidi et al. [ABB^{+}09] have proposed a formal framework for the dataflow pointcut, an original contribution by Masuhara and Kawauchi [MK03]. The dataflow pointcut describes where aspects should be applied based on the origins of data. The suggested use case is the detection of input-validation vulnerabilities, in which case a sanitizer, being the crosscutting concern, is applied.

```
pointcut sendIMEI(String o) :
  call(SmsSession.sendTextMessage(String)) && args(o)
  && dflow[o,i](call(String TelephonyManager.getDeviceID())
    && returns(i));
```

Listing 8: Dataflow pointcut for a sanitization task

A dataflow pointcut for a sanitization task could be defined as in Listing 8. The second line matches calls to the `sendTextMessage` method and binds the parameter string to variable `o`. The `dflow` pointcut is used to limit the join points to those whose parameter string was filled with the return value of `getDeviceID` at a previous join point. Masuhara and Kawauchi also provide an additional declaration form which makes it possible to specify explicit propagation of dataflow through external program parts. This proves useful when using third party libraries, native code or in any other situation where analyzable code is not available. The example from [MK03] demonstrates the syntax:

```
aspect PropagateOverEncryption {
  declare propagate: call(byte[] Cipher.update(byte[]))
    && args(in) && returns(out) from in to out;
}
```

Here, the system will assume that the return value from `Cipher.update` originates from its argument. As a result, if a string matches the dataflow pointcut, the encrypted string also matches the dataflow pointcut. This idea of explicit data flow propagation has not been adapted by Alhadidi et al. in their formal framework [ABB$^+$09].

The framework proposed by Alhadidi et al. uses dataflow tags which discriminate dataflow pointcuts. These are propagated statically between expressions to keep track of data dependencies. If an expression matches the pointcut of a dataflow pointcut, it is tagged. This tag is then propagated to expressions which depended on the original expression. The goal is to do as much of the tagging as possible statically, with dynamic methods being used for the remaining tags. This combination minimizes the necessary runtime overhead.

Monitoring properties like the one for SMS spamming (Listing 3), which requires the tracking of abstract state, cannot be supported by DFlow pointcuts, as such properties cannot be expressed as pure information flow.

6 PQL

Martin et al. have proposed a language called PQL (Program Query Language) in which queries about programs can be expressed declaratively [MLL05]. A PQL query matches sequences of method calls and field accesses in a target program. For every match, some user-defined Java code can be executed (optionally replacing the original method call in the target program). The following example calls a privacy checker (line 11) whenever private information (location, IMEI, etc.) is sent out in an SMS message:

```
 1 query main ()
 2 uses
 3   object * privObj, tainted;
 4 matches {
 5   privObj = TelephonyManager.getDeviceID()
 6     | LocationManager.getLatitude()
 7     | ... ;
 8   tainted := propagateStar(privObj);
 9 }
10 replaces sendTextMessage(tainted)
11   with checkAndSend(tainted);
12
13 query propagateStar(object * tainted)
14 returns object * y;
```

```
uses
  object * temp;
matches {
  y = tainted | { temp := propagate(tainted); y := propagateStar(temp); }
}

query propagate(object * tainted)
returns object * y;
matches {
  y = tainted.toString() | y = String.concat(tainted) | ... ;
}
```

Listing 9: Simple taint propagation in PQL

The `main()` query contains two matching-statements that must both apply for the overall query to match. Firstly, an object (we do not expect a concrete type here) must be obtained by a call to one of the listed functions, e.g. `TelephonyManager.getDeviceID()` for obtaining the IMEI. This object is bound to `privObj` and then followed through various propagation operators, such as direct assignment or string concatenation, using the patterns specified in the `propagateStar` query. If a match is found which is then used as a parameter for a call to `sendTextMessage`, this call is intercepted and the `checkAndSend` function is called instead. This function could then, for instance, ask the user for permission before actually sending the data.

This query shows that PQL supports modularity quite well. `propagate` is a second query that is used to filter executions based on what happens with the value previously matched for the `privObj` variable. The overall `main` query only matches if there is a suitable value for `privObj` returned by a call to one of the listed functions which then also matches the `propagate` query during further program execution. The `propagate` query can be reused in all queries that need to track information propagation.

The example above also shows that PQL supports recursive queries. String propagation may not occur at all (i.e., the variable of the ”get” call is used directly) or an arbitrary number of times. This is implemented using recursive references to the `propagateStar` query in which each reference corresponds to one `propagate` action. Additionally, one should note that matching in PQL works on black-box method calls and field accesses. Therefore, it does not matter whether the body of the called method is implemented in native code or in Java, or whether or not the source code is available, as long as the call site is located in analyzable Java code. In this respect, PQL is superior to many approaches.

7 Lessons learned / tradeoffs

We next briefly outline how well the four languages we studied address the requirements that we identified for the case of security monitoring of Android applications.

Reuse Among all surveyed languages, PQL is the only one that allows reuse of queries and have them call each other recursively. Query reuse is beneficial because it enables sharing of joint sub-queries. This decreases development time and also allows for a more efficient runtime evaluation, as shared sub-queries need to be evaluated just once. Other languages that support some form of query reuse include PSLang [Erl03].

Recursion Recursion increases expressiveness: Tracematches cannot effectively model information-flow properties because they can only reason about a fixed number of objects. JavaMOP generally shares this limitation; the only escape route is through resorting to plain AspectJ language features. Recursive queries in PQL avoid this problem through a renaming from actual to formal parameters, which effectively allows a query to reason about arbitrarily many objects.

Variable and member access Assume a policy that forbids the sending of any information about calendar entries marked as private. In such a case, the query language must provide a mechanism to restrict tracking to such entries `e` for which a predicate `e.isPrivate()` returns `true`. Tracematches, JavaMOP and Dataflow Pointcuts support such member accesses through a pointcut `if(e.isPrivate())`, PQL does not have such a mechanism. The only way to implement such a policy in PQL is thus to eagerly track *all* calendar entries accessed by the app. If a match is found, the handler can then check `e.isPrivate()` to see if `e` needs to be handled or not. This approach wastes runtime performance, as PQL must track objects which will never need to be handled later on. Furthermore, the filtering step is not part of the language, but of external code, making it harder to reason about the property actually being enforced. It also hinders the implementation of generic handlers (e.g. "ask the user for permission and then continue execution if granted") as the handler code must know the specific objects to filter them appropriately.

Customized monitor state Only JavaMOP allows queries to use custom data structures for tracking internal state. In Tracematches, Dataflow Pointcuts and PQL, matching is fully declarative. Customized monitor state increases expressiveness, which has both advantages and drawbacks. For instance, while the taint-tracking Tracematch from Listing 5 is very fragile with regard to code changes, the JavaMOP specification in Listing 7 is less so: using the custom data structure `taintedStrings`, the JavaMOP specification can track tainted objects recursively, and irrespective of any temporal order. Because any Tracematch specification can only track a fixed number of objects, this is a feature that Tracematch cannot support. The drawback is that custom data structures decrease readability and take away potential for static optimizations. After all, such data structures are outside the control of the specification-language compiler, which therefore cannot possibly reason about them. In the example, it may be beneficial to never add a certain string `s` to `taintedStrings` in the first place if it can be statically determined that `s` will never leak. An ideal language would provide fixed data structures that are expressive enough to support all common use cases.

Global, persistent state Unlike regular desktop programs, Android apps have to adhere to a specific life cycle in which they can be preempted by the virtual machine if the resources taken up by the app are required elsewhere. This life cycle requires monitors to serialize their state to disk, and resume monitoring when the app's execution is resumed. Another reason for allowing persistent state are properties that span multiple executions. For instance, a policy stating that an app may not send more than 10 SMS a month, no matter how often it was restarted, cannot be enforced without persisting state. None of the surveyed systems has automated support for such persistent state. It could, however, be supported through variable declarations whose values the runtime persists automatically. There are languages (e.g. ConSpec [AN08]) which support such a feature.

Implicit information flow None of the four languages are able to handle implicit information flow. In Listing 2, an attacker can fully reconstruct the IMEI, even though its value does not explicitly flow into any variable sent over the network. There is no direct path from a confidential source to an untrusted sink when considering just data flow. Furthermore, there is no trivial pattern (i.e., the IMEI may not flow into some *toBitStream* function) as the app is considered malicious and may thus employ any kind of code obfuscation. In the general case, the app's source code is not even known to the author of the security monitor. The one approach that would have potential to handle implicit flow is Dataflow Pointcuts. A pointcut `dflow[o,i]` states that data can flow from `i` to `o` without stating *how* data can flow. One could envision an implementation of Dataflow Pointcuts that included implicit flows in the pointcut matching process. For the concrete formalization by Alhadidi et al. [ABB+09], this is however not the case: their formal language does not even contain branching statements syntactically. For Tracematches, JavaMOP and PQL, implicit flows are impossible to handle simply because all approaches match against traces of *explicitly* mentioned events. A possible solution to this problem would be to make implicit flows explicit by exposing them as an event-like primitive `implicitFlow(o,i)`. This primitive would then be implemented similarly to Dataflow Pointcuts. Such a feature would in any case require static analysis beyond pure optimization and could not be handled solely at runtime since implicit flows need to take all possible executions into account and not just one concrete instance.

Primitive data types A similar problem regards the tracking of values of primitive types. All four studied languages can easily track objects, also through assignments between local variables, fields, arrays or even through reflection. This is possible because objects come with an identity, encoded in their object header. But primitive data types pose problems: Assume a user's phone book contains a phone number 12345, and an app sends the number 12345 to an untrusted server. But does this mean that the app is sending the phone number, or is it just coincidentally sending the same sequence of digits? In our running example, an attacker could exploit this ambiguity by obfuscating the code as follows:

```java
String myObfuscatedCopy(String in) {
  String resString = "";
  for (int i = 0; i < in.length(); i++)
    resString += in.charAt(i);
  return resString;
}
```

Here, `myObfuscatedCopy("12345")` will be equal to `"12345"` but not the same. Moreover, the name of the function `myObfuscatedCopy` is generally unknown. This problem can be solved by two different means. One possible solution is to regard all primitive types as objects, as, for instance, the case in Smalltalk [GR83]. Such an approach, however, could lead to significant performance degradation. A second possible approach is to use totally declarative information-flow specifications such as in Dataflow Pointcuts. In this case, the compiler would generate special code to track a primitive value's identity.

Native code Native code poses a quite similar problem. The contents of a method such as `native String myObfuscatedCopy(String in);` are not ready to be intercepted and analyzed by the security monitor. All four languages allow for manual specifications of the semantics of native calls, by instrumenting code at every call to those native methods. Manual specifications are only viable, however, if the native methods are known, e.g. because they are part of the standard library. Native code under control of the adversary cannot be specified. For such methods, a security monitor should make implicit worst-case assumptions ("The argument might flow into the return value."). None of the four studied approaches implements such a semantics.

Placement of sanitizers A sanitizer is a user-supplied function that converts a sensitive value into an innocent one, for instance by anonymizing, truncating or escaping data. Sanitized values do not require further tracking. It would therefore be beneficial if a query language would allow users to track all data "that has not been sanitized". In PQL, such behaviour could theoretically be emulated using negative patterns. The query should only apply if no sanitize method has been called:

```
query main ()
uses
  object * privObj, tainted;
matches {
  privObj = ... ;
  tainted := propagate(source);
  ~sanitize(tainted);
  conn.httpSend(tainted);
}
executes
  with Privacy.logViolation(source, tainted);
```

This concept however has several problems. Firstly, PQL can only apply negation to direct method calls or field accesses, not to queries. In the given example, `sanitize` must therefore be the name of a concrete method, no abstraction using query references is possible. Furthermore, the semantics of negated expressions can be quite surprising. It is important to place correct bounds on the scope of the negation. In our example, the query matches when the sink is called without the string having passed an anonymizer after its last propagation. In more complex flows, there might however be multiple positions where sanitization can occur. In such a case, the user would have to explicitly denote all of them in his query to avoid erroneous matches. The other three approaches show similar problems. Possible solutions include (1) fully automated sanitizer placement [LC13], (2) a scoping mechanism, allowing flexible negated queries with an intuitive semantics, and (3) a mechanism that would allow query programmers to restrict `propagate`-like queries within a set of user-defined sanitizer methods, i.e., define that the `propagate` query never matches inside a set of well-known sanitizer methods.

Support for proceed calls Tracematches and Dataflow Pointcuts support AspectJ-like `proceed` calls. This feature allows the specification to call the originally intercepted event with a possibly updated set of arguments. For instance, the Tracematch in Listing 5 proceeds with sending an SMS message, but first sanitizes the call argument, e.g. by removing or anonymizing private data. (More elaborate sanitizers are possible but outside the scope of this paper.) JavaMOP has no direct support for `proceed`, and neither has PQL. The latter requires users to explicitly enumerate the method calls that are intercepted and the handlers by which they are replaced (cf. Listing 9, lines 10–11). This is more verbose than just using `proceed` and hinders reuse.

Typing of the specification language A language to model (and enforce) security features needs convenient means of specifying data sources and sinks. We must be able to specify the default behaviour of operations that are not explicitly matched in a property. In an adversarial setting, this default would indicate that operations *propagate* tainted information, while if we would like to avoid too many false positives, we would assume information flow by default to *sanitize* data. Through a clear semantics, the language should communicate the intended behaviour clearly. Although a property could be encoded by just matching events and referring to global state (as in aspect-oriented programming in general or in the JavaMOP example in Listing 7), we consider an encoding on the *sequence of events* more readable than a solution where permitted/forbidden sequences of events are implicit.

The development environment for the specification language should adhere to the general concept of *static typing*. On the one hand, this will avoid load- or runtime issues where, for instance, operations in event handlers are badly defined (missing/wrong signatures). On the other hand, we do not seem to gain much flexibility through lax typing. PQL for example allows regular expression-based matches on method names *across classes* in a query, leading to a result of type `Object`. The handler method hence can only extract further information after inspection with `instanceof`.

For improved readability of specifications, (static) types provide a convenient means to *classify* abstract categories of data-types (e.g. "personal data", "security relevant"), and how operations on them propagate information. Any aspect-based approach may easily take advantage of this. Consider, for example, that the Tracematch specification in Listing 5 needs to track string operations for both the IMEI and the latitude in lines 2 & 3. Both could be subsumed in a case for arguments of type "security relevant" (again ignoring complications through primitive data types). Such a classification is implicitly carried out by means of state in the AspectJ/JavaMOP example in Listing 7.

Note that events do not only correspond to *functions* where the result has a particular property (derived from the parameters). In an imperative language, in addition to the return value, the classification of the callee may also change as a side-effect. For instance, a class could offer a `sanitize()` method. Whenever this method is called on an object, this object should no longer be tracked.

Discussion As explained above, all of the surveyed languages provide the one or other interesting feature that can be useful for security enforcement. However, no language is ideal, each one is lacking crucial features that are required to make enforcement truly reliable, in particular in light of obfuscated app code. Some of the points discussed above require tool support will require changes and additions mostly to the backend of the language implementations (native code, primitive types, implicit flow), while others do require substantial modifications to the languages' syntax and semantics.

While we studied a range of languages, there are other specification languages which we are unable to include here for reasons of brevity. Of particular relevance is ConSpec [AN08]. For the purpose of this study, suffice it to say that ConSpec is syntactically and semantically very close to Tracematches and JavaMOP, and presents roughly the same tradeoffs.

For practical usability, it must be sufficiently easy to express common security policies. Therefore, we argue for integrating the features discussed above into a single consistent language that allows flexibility where required, but also saves the user from details where unnecessary. The reuse of subqueries in PQL is a good example of providing such levels of abstraction. Furthermore, the languages need to be implemented efficiently and correctly. During our research, for instance, we found several issues with the PQL implementation, while there is no implementation for the DFlow pointcuts at all.

Ideally, users would be able to deploy their set of specifications onto their applications through the same well-known techniques that are already in use in existing monitoring frameworks.

8 Conclusions & Future Work

In this work we have investigated the suitability of using Tracematches, JavaMOP, DFlow Pointcuts and PQL for enforcing privacy-related security properties on Android apps. As we found, while each language has some interesting and useful elements, none of those languages fully address all requirements for this application area. An ideal language would define a minimal set of language constructs to solve the use cases we discussed. A flexible specification language for information flow properties allows reasoning about security guarantees and, together with instrumentation, the enforcement of such properties.

In the light of our observations, we would like to investigate whether we can also give recommendations with regards to API design. Usage of constructs that complicate a dynamic analysis could be discouraged by "marketplaces" like Google Play and the Apple App Store, which already run analyses on the code for quality assurance and to detect suspicious apps. For example, the `ContactsContract` content provider exposes the entire contents of the contacts. Here, for many applications being able to look up a particular entry e.g. by name or phone number might already be sufficient, without any need to explicitly crawl the address book.

For the examples in this study, we have mostly considered a “black and white” classification of data, i.e., no sensitive information may leak at all. When augmenting the analysis with user-defined data types, we might as well consider the case of *gradual* or *fractional* measures for quantifying information leakage and enforcing an upper bound on the amount of privacy-sensitive data leaving the user’s device. In this case, a user could define an individual balance between privacy needs and application requirements in cases where information leakage cannot be fully prevented without losing functionality.

Acknowledgements This work was supported by the Deutsche Forschungsgemeinschaft within the project RUNSECURE, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the bilateral DAAD/Research Council of Norway project “Runtime Verification for ABS Product Lines”.

References

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In [OOP05].

[ABB+09] Dima Alhadidi, Amine Boukhtouta, Nadia Belblidia, Mourad Debbabi, and Prabir Bhattacharya. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD ’09, pages 15–26, New York, NY, USA, 2009. ACM.

[AN08] I. Aktug and K. Naliuka. ConSpec–a formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, volume 197 of *ENTCS*, pages 45–58, 2008.

[And12] Android. Android Security Overview, December 2012. `http://source.android.com/tech/security/`.

[BGH+12] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. AppGuard — real-time policy enforcement for third-party applications. Technical Report A/02/2012, Universitäts- und Landesbibliothek, 2012.

[Bit12] Bit9. Pausing Google Play: More Than 100,000 Android Apps May Pose Security Risks, November 2012. `http://www.bit9.com/pausing-google-play/`.

[Cor12] International Data Corporation. Worldwide Quarterly Mobile Phone Tracker 3Q12, November 2012. `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37`.

[CR07] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA ’07, pages 569–588, New York, NY, USA, 2007. ACM.

[EGgC$^+$10] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *OSDI*, pages 393–407. USENIX Association, 2010.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[Erl03] Ulfar Erlingsson. The Inlined Reference Monitor Approach to Security Policy Enforcement. Technical Report 1916, Cornell University, 2003.

[FHM$^+$12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: an analysis of android SSL (in)security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[GCEC12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307. Springer, 2012.

[Goo12] Google Inc. Permissions, December 2012. `http://developer.android.com/guide/topics/security/permissions.html`.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[KB12] Eran Kalige and Darrell Burkey. A Case Study of Eurograbber: How 36 Million Euros was Stolen via Malware, 2012.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–354. Springer, 2001.

[LC13] Benjamin Livshits and Stephen Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, January 2013. To appear.

[MK03] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In Atsushi Ohori, editor, *Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In [OOP05].

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[OOP05] *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[ZZJF11] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on Android). In *Proc. of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107. Springer, 2011.