Constroid: Data-Centric Access Control for Android

Daniel Schreckling
Department of IT Security
ISL, University of Passau
D-94032 Passau, Germany
ds@sec.uni-passau.de

Joachim Posegga
Department of IT Security
ISL, University of Passau
D-94032 Passau, Germany
jp@sec.uni-passau.de

Daniel Hausknecht
University of Passau
D-94032 Passau, Germany
hausknec@fim.uni-passau.de

ABSTRACT

We introduce Constroid, a data-centric security policy management framework for Android. It defines a new middleware which allows the developer to specify well defined data items of fine granularity. For these data items, Constroid administrates security policies which are based on the usage control model. They can only be modified by the user of an application not by the applications itself. We use Constroid's middle-ware to protect the security policies, ensure consistency between a data item and its corresponding security policy, and describe how our prototype implementation can enforce a subset of possible usage control policies. In this way, our contribution shows how we overcome the rigid API-driven approach to security in Android. The structure and implementation of our framework is presented and discussed in terms of security, performance, and usability.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Control*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorised access*

General Terms

Design, Applied Security

Keywords

access control, usage control, Android, privacy

1. INTRODUCTION

Applications for smart-phones are mainly distributed using market platforms. Some of them perform shallow checks on the applications and verify that the developers comply with the terms of use. Other platforms omit such checks and rely on the option to withdraw applications in retrospect. Application distribution is mainly based on certificates generated by the market or by the developer. The operating system on the smart-phone checks these certificates and grants selected access rights requested by the application. This process can depend on the security features of the platform, the requested access, and on the decisions of a user.

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

1.1 Access Control in Today's Smart-Phones

Modern smart-phone operating system use widely deployed access control mechanisms and sandboxing techniques. These concepts are mainly based on process privileges (capabilities) or execution profiles [2, 8, 9, 11, 20]. They allow access to specific resources or processes. Once the access rights are granted to a particular process, it can perform the respective operations on the resource. Hence, access rights are selected for each application individually. Potentially malicious execution contexts are ignored. As a consequence, access rights granted to one application may be used by another application in a malicious execution context.

There are several approaches which try to improve this situation by addressing both, the granularity of available security policies, as well as their consistency. However, they mainly focus on security policies which control the processes and not the data processes operate on. As a consequence these security policies can only have low granularity. All data processed by the application is subject to the same security enforcement. Popular examples include access rights for address books. They often contain private, business, or public contacts. Once access to this resource is granted data with very different security requirements can be processed.

Further, the developer has to decide in advance in which context his application can or must run and which access rights might be required. Even approaches from the language-based information security and information flow area mostly employ static and process-centric access rules for data resources and follow the assumption: Get it all or nothing! However, in order to implement effective and usable security mechanisms for modern smart-phone applications this granularity is too coarse and too restrictive.

1.2 Contribution

We define a framework which allows for the secure management and partial enforcement of fine-grained and user-controlled access control policies in Android: Constroid. A middle-ware layer is introduced which abstracts from the actual storage of data on the physical device. Therefore, the developer has to define the semantic structure of the data his application processes. The middle-ware provides the appropriate access operations and offers fine-grained ac-

cess control for each individual item.

Constroid takes a first step to overcome the capability- and process-centric security model of Android. It addresses the elemental requirements of users who must guard the privacy of data but also need to process it with applications running in various contexts. Instead of specifying whether an application should be able to perform specific actions on resources we invert this approach and specify which actions should be allowed for specific data items. In so doing, we will be able to counteract applications or groups of applications which process data in ways not granted by the user.

We structure our contribution as follows: Before Section 3 outlines the basic access control model underlying Constroid, Section 2 explains the fundamentals of Android and reveals existing security problems. Afterwards, Section 4 shows how the usage control model is integrated in the Android system. Section 5 discusses the influence of Constroid on the Android system architecture in terms of security, performance, and usability. Finally, Section 6 compares it with related work and Section 7 concludes our work and outlines future research.

2. ANDROID FUNDAMENTALS

Android is – apart from some device drivers and the telephony stack – an open-source platform for mobile phones which was developed by Google. It builds on an embedded Linux and its libraries. They are exposed to application developers through the Android application framework implemented in the Java programming language. So, strictly speaking, Android forms a middle-ware [7] between a Linux system and the applications determined for the Android platform.

2.1 Application Architecture

Applications run on this middle-ware are executed in the Dalvik virtual machine (VM). It is register-based and optimised for running on devices with limited resources[1].

Applications are delivered in zipped Android Package files (`.apk` files). It contains the Java byte-code of an application, i.e. the Dalvik executable - also called `dex` file, resources relevant for the user interface or application specific data, as well as configuration files, e.g. a manifest. During installation, Android assigns each application a unique user and group ID.

Android applications support a dynamic and component-based structure. This allows for the reuse of existing components as well as a high adaptability. Four types of components are used to assemble applications: *activities*, *services*, *content providers*, and *broadcast receivers* [21]. Their execution is triggered by Android's application framework.

Application user interfaces are defined by *activities*. Data can be passed between activities by either passing parameters or by processing return values. Android suspends all activities which do not have the current focus and allows only one activity to gain keyboard focus.

In contrast, *service* components run in the background of an application. They are commonly used to process time-consuming or background tasks. Services can be started during device startup and update relevant databases, survey sensor inputs, etc. Remote procedure call interfaces are defined to interact or control such services.

[1]Available at `http://www.dalvikvm.com/` (August 2011)

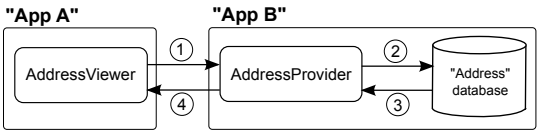

Figure 1: Data access using a ContentProvider

Broadcast receivers enable applications to receive messages broadcasted or sent specifically by other applications. They sign up for particular message types identified by labels, so called *intents*. This name is derived from the main usage of such messages: They express an intent to perform an *action* within a component. Broadcast receivers can also subscribe to general intents such as *"display the following image"* or *"show this location on a map"*. Therefore, broadcast receivers in combination with intents, are considered to form one of the most powerful features of Android.

The fourth component, *content providers*, offer an interface for sharing data. They basically offer the only way to make data accessible to other applications as data inherits its user and group IDs from the application they belong to. Requests to providers are based on common SQL queries. The actual data administration and response to queries is implementation specific to the individual provider.

2.2 Security Architecture

The Dalvik VM simplifies security enforcement in Android. The exchange and reception of intents, the call of protected APIs, the use of content providers etc. can be monitored centrally.

For this purpose, every application must provide a manifest. By defining specific permissions an application can request the right to execute specific API functions, prevent the launch of activities by other applications, control the reception or broadcast of intents, can control whether an application can bind to a service, or whether the application can access specific hardware or memory resources. If an application does not stick to its requested permissions and tries to access a resource it does not have the right for, the Android runtime environment throws an exception, notifies the user, and terminates the execution of this application.

Permissions are granted at install time of an application. Depending on the protection level, this is done automatically by the system or if required with user interaction. If at least one of the requested permissions is not granted, the installation of an application is aborted. Permissions are granted statically, i.e. they cannot be revoked or added later.

Android offers several APIs to store application data in files or databases. Every application possesses a separate data directory to store this data. To ensure access control on these files, Android makes use of the traditional access control mechanisms deployed in Linux systems using the unique user and group identifier assigned to an application. To exchange data with other applications, Android applications must define content providers (see Section 2.1). Access rights to these providers are defined statically in the manifest. Another option to share data are intents. As explained above, access to these messages must also be defined in the applications' manifest.

Assume a common situation: Application *App A* wants to read data managed by application *App B*. Figure 1 depicts the single steps required to query data from *App B*.

Activity *AddressViewer* displays address information in *App A*. *App A* does not store any address information but uses the data managed by *App B*. Android forbids *App A* to directly access this data. Therefore, *App B* implements the ContentProvider *AddressProvider* and protects it with a permission of level *dangerous*. Permissions of this level must be approved by the user at install time. Supposed *App A* requested this permission and it was granted by the user.

To display an address a, the activity of *App A* uses the ContentProvider to query (1) address a. Android's security monitor verifies whether this query is allowed. The *AddressProvider* uses the API to pass the received query (2) to the SQLite database. The SQL query operation returns a DB cursor that contains address a (3). This cursor is returned to the *AddressViewer* (4). The Activity of *App A* can now read the address data and display it.

2.3 Security Issues and Deficiencies

Android also allows the definition of application specific access control mechanisms. Applications can delegate the right to access a particular resource. This resource must be specified using a URI. An intent sent to the delegated application equipped with the correct permissions finally grants the access. In this way, an image viewer is able to display email attachments although it is not able to access the whole mailing database.

With this mechanism the individual developer decides during the design and implementation phase, which security policies fit the user and the application. As a result an application requesting data always depends on other applications which enforce the access permissions. However, stand-alone applications often require direct access to various resources, e.g. the address database, the media database, etc. To respect all existing applications and applications potentially implemented in the future, the developer would have to implement a multitude of access control functions or content providers.

This reveals another deficiency of Android's permissions management: permission assignment. During installation time, the user must decide which permissions an application should possess. Effectively, he must consider which permission could compromise which data. However, if the user does not accept the demanded permissions the application will not be installed. Revoking these permissions further requires the de-installation of the application, their modification and re-installation. As a consequence, the rights to application assignment – regardless of the system state, other applications, or the security requirements of the user – has to be advanced to implementation time, to the developer.

Even worse, granting permissions to different applications may implicitly compromise private data. By granting access to other applications the user builds some kind of trust chain which may not fit all data items this chain can process. Thus, the user grants – mostly unknowingly – implicit access rights to applications. An access control system which is simply based on assigning capabilities to processes will not solve this problem. Systems based on data-usage decisions are required and thus systems which can provide data security policies.

3. SYSTEM MODEL

We introduce the model which inspired the design of Constroid: Usage Control with *A*uthorisation, o*B*ligations, and *C*onditions ($UCON_{ABC}$). Based on this model this section briefly outlines which policies Constroid is able to define on data items.

3.1 Usage Control

Constroid is based on the family of $UCON_{ABC}$ models as introduced in [18]. It extends the classical notion of access control and allows for modelling of numerous usage policies. While it is able to describe classical models such as DAC, MAC, and RBAC, it also meets the requirements of common DRM systems. With the dimensions of *Authorisations*, *Conditions*, and *Obligations*, it addresses Constroid's need for expressive policies able to describe how personal data should be safeguarded against potential misuse.

Subjects in Constroid are the operating system or Android processes. This deviates from the classical UCON model where a subject represents an individual human being. Although the user of a smart-phone may authenticate to the phone, we assume that there is basically one active user executing processes in different contexts. Thus, *subject attributes* can be any characteristic describing a process, e.g. its user and/or group ID, its state, etc.

Through Constroid, subjects may obtain rights (see next subsection) to operate on objects. The latter are represented by so called *data items* (see Section 4.2). They are also associated with *attributes* which influence the decision about the rights a subject may obtain for an object. Thus, such attributes include the ownership of data items, i.e. which process currently owns the item, which class it belongs to, e.g. whether it is a contact item with a set of additional information such as name, first name, address, etc. In general, object attributes are associated with the object itself and express certain characteristics.

Rights are privileges that can be assigned to a subject to perform specific operations on an object. Constroid, supports the basic rights *create*, *read*, *update*, and *delete* (CRUD). They are enforced when accessing the data stored in secondary memory. This basic set can be extended to specific API functions if feasible mechanisms would be integrated into the VM. Rights are assigned to a subject using the *decision function*. It uses subject and object attributes as well as authorisations, obligations, and conditions to determine the set of appropriate rights.

In Constroid, *authorisations* (A) are functional predicates which have to be evaluated for usage decisions. They are used to decide whether a subject is allowed to perform the request operations on an object, i.e. whether it is privileged to obtain the appropriate rights for an object. UCON distinguishes pre-authorisation ($preA$) or ongoing-authorisations (onA). Apart from $preA$, Constroid also allows for onA to support policies which contain temporal or spacial constraints.

Obligations (B) in Constroid describe functional predicates which verify that a process has performed before ($preB$) operating or performs during (onB) the operation on an object. An example for a $preB$ predicate is a list of API calls a subject (process) has to fulfil before transmitting a data item to a specific server. An onB predicate could verify that a process maintains an encrypted connection during its execution to exchange particular data items.

Finally, *conditions* refer to environmental or system-related factors. By evaluating states of the environment, e.g. GPS location, speed, etc. or by analysing states of the system,

e.g. system time, network connectivity, etc. the model can provide information which is used to decide which rights to assign to a subject. More specifically, conditions can be used in the subject or object attributes to control the rights assignment.

3.2 Policy Structure

Constroid defines policies exclusively for data items or resources generating data. The user defines in which way data must or must not be used. As Constroid's main task is to securely manage security policies our prototype is currently limited to a subset of policies $UCON_{ABC}$ can express. Further, it is currently limited to the initial access to data. Therefore, the user is restricted to the following subject and object attributes.

Subject attributes can use Android's user and group identifier of the process accessing the data. Later versions of this system will also support particular states or traces of states. Object attributes include the time of use, the processes allowed to access an item, and a geographical location in which the item can be accessed. To specify access rights the user is limited to CRUD operations as mentioned before. These dimensions can be combined arbitrarily, i.e. complex policies can be assembled by using the binary logical operators `AND` and `OR`, and the unary logical operator `NOT`. Empty policies, i.e. policies which do not specify any attributes or rights, are most restrictive. Any access to a data item which is associated with an empty policy results in a negative authorisation decision: Access is denied. Any data item generated by a system resource is assigned a default policy associated with this resource.

The specification of the data policies is accomplished by using a simple user interface which allows the definition of simple policy constraints as well as their combination.

4. INTEGRATION AND OPERATION

This section first sketches the architecture of Constroid, gives a short example of its basic operation and describes the single components in more detail.

4.1 Overview

The enforcement of our model is based on a reference monitor concept. To guarantee that no application obtains unauthorised or unmonitored access to data items, we introduce a middle-ware layer between the Android application layer and the APIs enabling data access (see Figure 2).

We assume that Constroid is the only system component in Android which offers an interface to access data. This interface (a) is called the Application Content Provider (ACP) and represents our reference monitor. It allows for CRUD-operations on data objects. Every application is assigned an individual instance of an ACP. It uses a Data Manager (DM) that administrates all data processed or required by an application (b).

Hence, the ACP implements the Policy Enforcement Point (PEP) and allows data access only if the Policy Decision Point (PDP) returns a positive check result. The PDP as well as the other components required for policy handling, i.e. the Policy Administration Point (PAP) and the Policy Information Point (PIP), are implemented in an extra Android Service (c), the Policy Handler Service (PHS) which is integrated in the Android system image. As a consequence, every ACP requires a connection to the policy handler to be

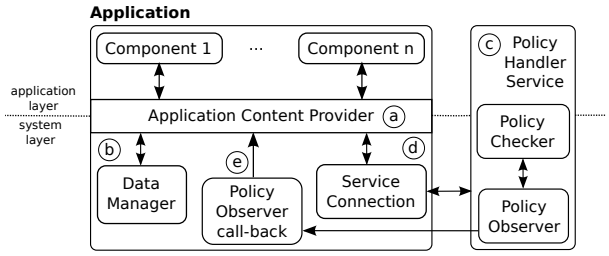

Figure 2: Basic framework design.

able to request policy checks (d).

The PDP performs one-time checks on data item access policies and can monitor open data connections such as a database cursor or file streams. The ACP can pass a call-back interface (e) to the PHS to get notified as soon as the conditions defined in an monitored policy are no longer met.

Remember the simple example from Section 2.2. Suppose that data access in *App A* and *B* is implemented using Constroid. Additionally, assume that data consist of the data items *name*, *road*, and *city*.

Figure 3 shows the single steps required to query a data item. In the first step the *AddressViewer* queries the *AddressProvider* for address a (1). Instead of accessing the address DB directly the *AddressProvider* must now use the interface of the ACP to query the appropriate data (2). The ACP then uses the DM to request the desired address a (3). To make an authorisation decision the PHS is called (4). It checks the access permissions of the items of a: *name*, *road*, and *city*. If all access policies are satisfied, a database cursor is returned to the *AddressProvider* (5) which is then forwarded to the *AddressViewer* (6). The requester can read and display a.

4.2 Data Encapsulation

Content Providers, direct URI access, and the Android API in general allow arbitrary manipulation of data. To enforce access control policies on a fine-granular level, i.e. for single data items, Constroid must describe how a *data item* is defined. It must ensure that all operations performed on a data item can be monitored, guarantee that a policy is uniquely associated with its data-item, and that it is updated consistently.

For this purpose, Constroid forces the developer to define an XML file called `Datastructures.xml`. It is similar to the `AndroidManifest.xml` file which must be defined for every application. In this file the developer specifies how he intents to structure the application data. A flat and recursive structure as sketched in Figure 4 is used. A specification

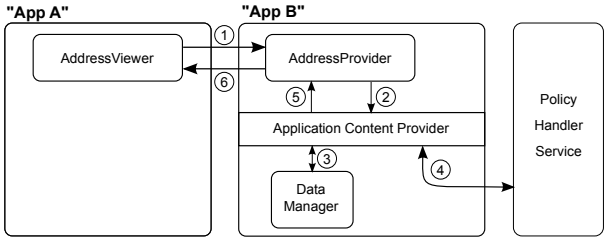

Figure 3: Example for Data access with Constroid

```
<datastructures>
  <structure name="Contact">
      <item name="name" type="string" />
      <structure name="Address">
        <item name="road" type="string" />
        <item name="city" type="string" />
      </structure>
  </structure>
<datastructures>
```

Figure 4: Example for a data structure definition.

consists of a `structure` which consists of single data `items` or other `structures`. So, data structures are templates for data instances and consist of at least one data item and zero, one, or more substructures. Data items are the smallest elements of an data object and are assigned a name and type. Figure 4 shows the specification of a contact structure which itself contains an address structure.

The DM uses the definition in `Datastructures.xml` to generate the appropriate storage structures used to store and manage instances of data objects on the device. Further, the DM is part of the reference monitor. This design decision was made because the system must have full control over the usage of the DM and thus the application data. Further, data structures are defined individually for every application. So, integrating the data manager in the reference monitor eases data handling.

Finally, as the DM becomes an intrinsic part of an application the DB used for administrating the data of the DM can be stored in the application directory with the UID of the application. In this way the set of all data items is protected by the existing security mechanisms of Android.

4.3 Policy Enforcement

Isolating raw data through a well defined API is one measure to enable fine-grained access control. The second step associates security policies with all data items. A policy is a set of constraints that specifies the context that must be given for granting access to a particular data item. The policy framework required to administrate, evaluate, and enforce these policies will be described in this section.

4.3.1 Policy Administration Point

The PAP of Constroid is the PHS. It offers a user interface which allows for the consistent administration of access policies for every application. It also manages the default policies for data items generated by applications. The security architecture of Android forbids local implementations of the PHS as it requires access to the security policies of all applications. Therefore, it is implemented as an Android service and is integrated in the system image.

The PHS further implements the policy decision point (PDP) and the policy information point (PIP). This prevents potential vulnerabilities because access policies never leave the PHS and are fully protected by its application sandbox. Additionally, the integration of PDP and PIP increases the framework performance as it avoids additional IPC calls.

4.3.2 Policy Decision Point

The policy decision point consists of two components: the *Policy Checker* and the *Policy Observer*.

For simple requests, the *policy checker* (PC) performs one-time checks on access policy sets and makes authorisation decisions. The policy handler service queries the access policies using the PAP and passes them to the PC.

If an application component wants to access a set of data items, e.g. data in the column of a DB, all associated access policies must be satisfied. If the access to one of the data items is not permitted, access to the whole set is forbidden. Therefore, based on the requested data, the PC first computes a set policy. It unifies the constrains of all policies specified for the queried data items. This union is used for making the authorisation decision. For the evaluation of the policy the PC uses the PIP, e.g. to obtain location information.

For entities which allow long-time access, such as database cursors or file streams, the *policy observer* (PO) is required. It allows the interruption of an open I/O-connection as soon as its access policy is no longer satisfied. For this purpose, the PO re-evaluates data policy constraints for policies registered with the observer if the corresponding context information was updated. Registered policies are observed as long as their constraints are met. If a constraint check fails, the corresponding access policy is unregistered from the PO and the application that required the policy checking is notified via the PO callback component. In case an application closes a database cursor or file stream the PHS un-registers the appropriate access policy as well.

4.3.3 Policy Information Point

The PIP is implemented by the context information provider. It offers an interface that allows to retrieve context information required for checking the constraints of a policy. It further allows POs to register for context information updates. In case the PIP can not obtain specific context information, e.g. it can not determine GPS data, it will inform any registered PO.

4.3.4 Policy Enforcement Point

Constroid enabled application components must use the ACP to access data items. As it uses the DM to access data, it is located between the application components requesting data access and the DM (see Section 4.2). This position allows for complete control over the data access.

The PEP holds a connection to the policy handler service. If this connection cannot be established for any reason, no authorisation decision can be requested and thus every data access is denied by default.

When data access is requested the ACP identifies the set of affected data items and requests an authorisation decision from the PDP. Depending on this result the access to the data item set is granted to the application component or it is denied. For policies with spacial or temporal constraints, the ACP can instruct the policy handler service to observe access policies after an initial check by the PC. The PO will notify the ACP as soon as one of the constraints in the policies associated with the accessed data items is unsatisfied. To enforce the policies the ACP will then close any handle which is used to access the data.

5. EVALUATION

This section discusses the usability and security of the middle-ware introduced with Constroid, its performance overhead, and its impact on the usability.

5.1 Security Analysis

The PHS must be publicly available. This implies that every application component can access this service and try to manipulate the policy database. In this section we consider a malicious application not able to by-pass the established Android security mechanisms and which can only access data by means of the ACP. We identified the following possible attacks the application can perform against Constroid to gain unauthorised access to data, i.e. access to a data-item which conflicts with its associated security policy.

The ACP relies on the PDP implemented in the PHS. A denial-of-service attack may *crash the PHS*. However, if no connection to this service can be established all access to data objects is denied by default. Instead of crashing the PHS an attack may try to *replace it with another service bypassing the PHS*. As the PHS is part of the Constroid system image it is protected by the certificate of the manufacturer. Thus, the PHS can only be updated or replaced if the new application was signed with the same key.

Instead of manipulating the PHS a malicious application may try to *guess identifiers of observed policies* and try to un-register them. In this case, Constroid would deny further access to the data item. However, the operation to un-register can only be called for data-items an application owns. So, a malicious application does not obtain unauthorised access. It can only perform this denial-of-service attack against itself. Another attack may aim to *create new policies for arbitrary items* using the PHS. In this case a query tries to generate a policy for a non-existing item. This policy will be generated but is never used. If a policy already exists, the request to create a policy will be ignored. Existing policies are not replaced. This also holds if the PHS needs to install a new policy for a data-item just generated by the application. If this policy already exists, it queries the user to change the policy of the item. Only after this feedback the item can be used in the framework. Also the *deletion of policies* is a potential attack. However, this restricts access to an item and results in a pure denial-of-service attack. Unauthorised access is not possible.

Finally, an application may try to *manipulate the PIP* of Constroid to annul spacial or temporal policy constraints. However, the PIP uses system services to retrieve its information and is encapsulated in the PHS. Apart from a physical manipulation which can not be performed by an application but a real user, a manipulation is not possible.

Thus, apart from denial of service attacks, a malicious application cannot successfully manipulate the policy system to gain unauthorised access using the attacks listed above.

Finally, the security of the information protected by Constroid depends on user decisions. Of course, we cannot prevent the user from being incautious during policy definition. However, policies are specified using temporal or spacial contexts on data items. In this way, Constroid allows for comprehensible interactions with the security enforcement.

5.2 Performance

We tested our prototype system on an HTC Desire. It is equipped with a 1 GHz ARM processor, 576 MB RAM, and a 16 GB microSD card. The device runs Android 2.2 without manufacturer additions. For our experiments we implemented a simple application. It generated 1000 contact instances of the data structure defined in Figure 4. The content of these items was random. Similarly, the security policies for these items were randomised time constraints. Each contact was inserted into an empty DB using the API offered by Constroid. We exclusively used inserts for our benchmark as the overhead for this operation is maximal. This experiment was repeated 50 times and showed a mean execution time of 1933 ms with an average deviation of 55 ms and a 99% confidence interval of 19.9 ms. We compared our results with a standard Android 2.2 compilation. The test application performed equivalent inserts into an SQLite DB using the original Android API. This experiment was also repeated 50 times and showed a mean execution time of 1614 ms with an average deviation of 87 ms and a 99% confidence interval of 31.8 ms. Thus, our prototype implementation shows a runtime overhead of 21-22%.

The storage overhead induced by the security policies can not be stated precisely. First of all, we currently do not have exact usage data. Thus, it is hard to claim that policies will have an average complexity. Further, it is hard to estimate how many different policies a user would define. He might prefer different security contexts, such as work or home, and assign these *context-roles* to his data. This would again decrease the memory overhead. We are also not sure how fine-granular a developer may structure application data. The higher the granularity a developer offers, the higher the storage overhead. Finally, extending the dimensions usable for policy definitions may additionally increase the storage overhead.

5.3 Usability

Yet, no specific user interface has been implemented. Constroid only provides the interfaces to manipulate data offered by the ACP. However, we think that the actual security policies required for Constroid can be abstracted in a feasible way. In particular, spacial and temporal attributes can be visualised easily and comprehensible. Also the access of other applications to local application data or the distribution of this data to other remote locations can be represented in a feasible way. We further think that policies can be modified on-demand, i.e. if a usage constraint is not satisfied an appropriate dialogue will help the user to understand the consequences of his policy adjustments.

In the end, it is not clear whether our policy system will overwhelm the user or empower him to transparently and comprehensibly guide new data-centric security systems.

5.4 Application integration

Constroid must ensure that it can monitor any access to a data item. Android, however, offers numerous ways to access and manipulate data. Even if we would be able to wrap all interfaces which allow data manipulation, we would not be able to derive the precise data item manipulated by a certain API call, e.g. in an SQL query or a binary file access.

Therefore, the design of Constroid must limit the freedom of the developer. He must first model the application data. This model is defined in a separate XML file. While this may appear to imply extra work, this step is commonly required by every developer who must administrate data.

Further, during every data access the developer has to indicate which data item he wants to access. This implies the use of interfaces specifically defined for Constroid. The administration of the data, e.g. whether it is stored in binary format or in a database, is managed by Constroid. Thus, the developer can not implement efficient routines feasible

for the type of data they process. However, the operating system may provide optimised methods for specific types.

In case a developer decides not to use the Constroid framework, he can *circumvent* this security system during application design. As an example, he may declare one binary file in his data model. Operating on this binary file is then up to him. In this case, the user can only specify one policy for the binary. Specifying policies for single data items will not be possible and protection of private data is reduced to the current security mechanisms of Android. We assume that a security aware user will decide not to use such applications.

6. RELATED WORK

Our work encompasses purely theoretical concepts such as in [22] and focuses on the realisation of these models in the smart-phone OS Android. Therefore, this section focuses on related work which deploy practical implementations.

To our knowledge, the *Core Data* for iOS [1] is the only framework which uses some type of middle-ware layer to structure application data. However, iOS only uses this concept to tag data objects with meta data which support internal services, such as the finder. Apart from simple confidentiality flags for address or calendar entries which are often enforced on the server, fine granular security policies for data do not exist in widely spread smart-phone OSs.

Further, it is hard to compare our work with systems such as Panorama [23], Trishul [13] or TaintDroid [6]. While Constroid securely manages security policies, these systems mainly focus on the tracking of properties. In particular, Panorama and TaintDroid do not allow for the dynamic specification of data security policies. However, they are related as their precision can be improved when coupling their taint tracking features with more precise, general purpose policy management systems such as Constroid. Trishul and the VM based flow control system described in [12] take a step towards this approach and allow to process security policies tagged to data. However, the authors do not describe how these policies are specified and administrated in a secure way. Further, the employed policies only operate on the file system or resource level. The mechanisms with which items are linked to a particular and user-defined security policy remain open.

Porscha [16] and T-UCON [14], DRM frameworks based on TaintDroid and Trishul respectively, allow content sources to define security policies. The basic idea to protect individual data is close to our approach. However, these systems focus on DRM protected content, Constroid aims at any type of data processed by applications. Secondly, Porscha and T-UCON only mediate the exchange of data between applications, Constroid controls the access to data by means of a reference monitor. Further, Porscha and T-UCON try to safeguard specific data against unauthorised use on any platform. The approach of Constroid is to shield personal data from illegal use by any application.

Saint [17] is a framework that extends Androids permission mechanism by allowing to define rules for granting permissions. It defines application-centric rules. These constraints are static and enforced at runtime. Thus, it is again the task of the developer to decide on the end user's privacy. Users can only disable or re-enable an application rule.

A similar framework is Apex [15]. It also extends Android's permission mechanism. It allows the user to grant or deny a subset of the permissions requested by an application. The permissions of an application can be reconfigured at any time. In contrast to Saint, the user gets control over the granted permissions instead of the application developer.

However, the permissions defined in Saint and Apex only allow access to complete resources and remain application-centric. This may not be desired in all application contexts. Therefore, these extension are insufficient as they give not enough control over the accessibility of specific data objects.

The same holds for Kirin [7]. It is a security service which analyses the configuration of an Android application and detects potential security issues. It primarily analyses the permission requests and compares them with the security policies set for a device. As Kirin is also based on declarations and requests in the Android manifest it can still not prevent or detect unauthorised data access. Its view on data is too application-centric to distinguish unauthorised requests within a resource, such as an address book.

The European project S3MS generated a series of publications [3, 4, 5, 19]. They implement the concept of security by contract in slight variations and on different platforms. Contracts describe a policy according to which an application should behave. Obviously, this approach is application-centric. The use of the application and its specific operations on data have to be known during development. Security policies which define the security requirements of a user for a single item can not be specified.

Hence, Constroid clearly distinguishes from other related work by deploying data-centric and user-defined, dynamic policies, its application independence, and the finer granularity of access policies.

7. CONCLUSION AND FUTURE WORK

In Android, large attention has been given to the secure communication between applications [8] as well as to the secure usage of system specific APIs. A large variety of permissions can be used and grouped to protect applications. Several approaches refined these security mechanisms to mediate their deficiencies. Surprisingly, apart from highly specific solutions for DRM content, none of these approaches tried to refine the security policy management for data and to complement the traditional security management inherited from the Linux system Android is based on.

To our knowledge, Constroid is the first system for a smart-phone operating system which allows for the definition and storage of data-centric, fine-granular security policies. Data items can be tagged with data security policies which are based on the $UCON_{ABC}$ model. We developed a prototype system for Constroid. It enables policies which respect attributes such as the application identifier, the type of operation during initial access, time, and location.

Our approach clearly differs from established permission management systems in almost all smart-phone operating systems. We focus on a data-centric instead of an application- or process-centric permission system. This difference is crucial, as it constitutes a first step towards breaking with the widely established paradigm to define permissions during distribution and development of applications. Current and future application architectures, e.g. mashups or peer-to-peer systems, require more user-centric approaches. Static, application-centric, *one-for-all permissions* are simply not feasible for applications which can be used in completely different contexts, processing the same user data. Thus, despite the restrictions Constroid imposes on the developer

and in the face of the introduced overhead we think that Constroid takes an important first step towards an alternative to currently established permission systems.

Constroid further paves the way for new security mechanisms. Future work will investigate techniques which use the security policies stored in Constroid to inspect the information flow of single applications or their possible composites. These techniques can enhance and exceed existing dynamic enforcement mechanisms [6, 10, 14, 16]. We strive for the development of static analysis and code inlining techniques for Dalvik executables. Their application will enable us to include inline reference monitors to enforce security properties of information flows which are compliant with the security policies specified by the user using Constroid.

In order to apply Constroid in this domain we will first conduct a case study with a Constroid version able to enforce the full expressiveness of usage control, i.e. based on the policies stored in Constroid it will track data and control its usage. Datastructures with good performance will have to be developed to reduce the number of database queries and to allow for an efficient coupling between the Dalvik VM and the Constroid policy administration.

8. REFERENCES

[1] Apple Inc. Core Data Tutorial for iOS. Available at: http://developer.apple.com/library/ios/. June 2011.

[2] Apple Inc. Security Overview. Technical report, Cupertino, CA, USA, July 2010.

[3] A. Castrucci, F. Martinelli, P. Mori, and F. Roperti. Enhancing Java ME Security Support with Resource Usage Monitoring. In *10th International Conference on Information and Communications Security*, volume 5308, pages 256–266, Birmingham, UK, October 2008. Springer-Verlag Berlin Heidelberg.

[4] G. Costa, A. Lazouski, N. Dragoni, R. Saadi, and D. Ingegneria. Security-by-Contract-with-Trust for Mobile Devices. *Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications (JoWUA)*, 1(4):75–91, December 2010.

[5] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, January 2008.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI 2010*, pages 1–6, Vancouver, BC, USA, October 2010. USENIX Association.

[7] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, 2009. ACM Press.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security & Privacy Magazine*, 7(1):50–57, January 2009.

[9] C. Heath. *Symbian OS Platform Security, Software Development Using the Symbian OS Security Architecture*. John Wiley & Sons Ltd., 2006.

[10] J. Liu, M. D. George, K. Vikram, L. Waye, and A. C. Myers. Fabric : A Platform for Secure Distributed Computation and Storage. In *22nd ACM Symposium on Operating Systems Principles*, pages 312–334, Big Sky, MT, USA, October 2009. ACM Press.

[11] Microsoft Corporation. Windows Phone 7 Security Model. Technical report, December 2010.

[12] S. Nair, P. Simpson, B. Crispo, and A. Tanenbaum. A Virtual Machine Based Information Flow Control System for Policy Enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, February 2008.

[13] S. Nair, P. Simpson, B. Crispo, and A. Tanenbaum. Trishul : A Policy Enforcement Architecture for Java Virtual Machines. Technical report, Vrije Universiteit, Amsterdam, Netherlands, 2008.

[14] S. Nair, A. Tanenbaum, G. Gheorghe, and B. Crispo. Enforcing DRM policies across applications. In *Proceedings of the 8th ACM workshop on Digital rights management - DRM '08*, page 87, New York, New York, USA, 2008. ACM Press.

[15] M. Nauman, S. Khan, and X. 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*, pages 328–332, Beijing, China, 2010. ACM Press.

[16] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy Oriented Secure Content Handling in Android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, New York, NY, USA, December 2010. ACM Press.

[17] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *2009 Annual Computer Security Applications Conference*, pages 340–349. IEEE Computer Society, December 2009.

[18] J. Park and R. Sandhu. The $UCON_{ABC}$ usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, February 2004.

[19] P. Philippaerts. *Security of Software on Mobile Devices*. PhD thesis, Department of Computer Science, Faculty of Engineering, Leuven, Belgium, October 2010.

[20] Research in Motion Ltd. BlackBerry Enterprise Solution, Security Technical Overview for BlackBerry Enterprise Server Version 4.1 Service Pack 6 and BlackBerry Device Software Version 4.6. Technical report, Canada, March 2009.

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

[22] C. Schaefer. Usage Control Reference Monitor Architecture. In *Third International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing (SecPerU 2007)*, pages 13–18. Ieee, July 2007.

[23] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 116–127, New York, NY, USA, 2007. ACM Press.