Kynoid: Real-Time Enforcement of Fine-Grained, User-Defined, and Data-Centric Security Policies for Android

Daniel Schreckling, Joachim Posegga, Johannes Köstler, and Matthias Schaff

Institute of IT-Security and Security Law
University of Passau, Germany
{ds,jp}@sec.uni-passau.de, {koestler,schaffma}@fim.uni-passau.de
http://web.sec.uni-passau.de/

Abstract. We introduce Kynoid, a real-time monitoring and enforcement framework for Android. Kynoid is based on user-defined security policies which are defined for data-items. This allows users to define temporal, spatial, and destination constraints which have to hold for single items. We introduce an innovative approach to allow for the real-time tracking and enforcement of such policies. In this way, Kynoid is the first extension of Android which enables the sharing of resources while respecting individual security policies for the data-items stored in these resources. We outline Kynoid's architecture, present its operation and discuss it in terms of applicability, performance, and usability. By providing a proof-of-concept implementation we further show the feasibility of our framework.

Keywords: Android, security, security policies, information flow.

1 Introduction

The distribution of Smartphones to employees becomes more and more interesting for companies. They enable unified and simplified communication as well as permanent reachability. These companies also tend to weaken their traditional security requirements by weakening guidelines and restrictions on these devices to avoid that users replace their business Smartphone and use other means of communication for private purposes. At the same time, modern platforms have seen a tremendous increase in innovative applications. They provide easy access to web and cloud services and support the user in their daily activities. This trend is enforced by simple and handy APIs which inspire private application developers. As a consequence, application markets have become very popular from a developer as well as from a consumer perspective.

1.1 Information Processing in Today's Smart-Phones

This development needs support by feasible Smartphone operating system platforms. Among other requirements, the ability to share information between applications, is essential. However, the security mechanisms which are currently

© IFIP International Federation for Information Processing 2012

available to control access and enforce specific security requirements are not suitable and insufficient.

Assume a very common situation in which a company distributes Smartphones to their employees. The company allows the private use of the cell phone including the installation of applications. Thus, the resources of this Smartphone, e.g. the address book, the file storage, or the browser history and bookmark database, will contain private as well as business data-items. From a privacy and from a confidentiality point of view, this is an unbearable situation in particular if we consider currently available security mechanisms protecting this information. Modern smart-phone operating systems use widely deployed access control mechanisms and sandboxing techniques. These concepts are mainly based on process privileges (capabilities) or execution profiles [1,7,8,11,18]. They grant coarse grained access to resources or processes. Once the access rights are granted to a particular process, it can perform the respective operations on the complete resource. Hence, access rights are selected for each application individually. Potentially malicious execution contexts are ignored.

Several approaches improve this situation by addressing both, the granularity of available security policies, as well as their consistency. However, they also focus on process centric security policies. As a consequence these security frameworks cannot have the level of granularity required to express security policies for individual data items. All data processed by the application is subject to the same security enforcement. Thus, it is not surprising that applications often synchronise even private or confidential contacts with inappropriate infrastructures. In our setting, an employee may install an application for a social platform. If the application gains access to a shared resource, e.g. the address book, it can accidentally start to synchronise it with the social platform, including potentially critical data.

1.2 Contribution

We define a framework which allows for the real-time tracking of fine-grained and user-controlled security policies in Android: Kynoid. We enhance the taint-tracking system TaintDroid [5] and distinguish not only between coarse grained data-sources but introduce security policies for individual data-items. In this way, a user can dynamically specify location and temporal constraints where and when data can be processed and restrict the destinations to which data-items are allowed to be distributed. Kynoid enforces these individual security policies at data sinks. In so doing, Kynoid overcomes the capability-, process-centric, and coarse grained security models implemented in todays' Smartphone operating systems. Through the implementation of a first unoptimised prototype, Kynoid further shows that it is not justified to assume that the tracking of complicated security policies during runtime is very expensive and therefore not suitable. Thus, our contribution introduces the first approach which is able to define security policies for single data items and which allows their dynamic enforcement.

We structure our contribution as follows: Section 2 explains the fundamentals of Android required to understand the work presented in this contribution. Afterwards, Section 3 gives an informal overview of our solution before Section 4 introduces the theoretical and technical details of Kynoid. Section 5 discusses the influence of Kynoid in terms of applicability, performance, and usability. Finally, Section 6 compares it with related work and Section 7 concludes this work and outlines future research.

2 Background: Android and TaintDroid

Android is a popular software stack for mobile phones developed by Google. Apart from some device drivers and the telephony stack, Android is open source. It builds on an embedded Linux and its libraries which are mainly programmed in C/C++. Android allows the use of these libraries through an application framework which is implemented in the Java programming language. Therefore, Android can be considered as a middle-ware [6] between a Linux system and the applications programmed for the Android platform.

2.1 Dalvik Virtual Machine

Applications intended to run on Android must also be written in Java and are compiled into custom byte-code. It is executed by the Dalvik virtual machine (VM) which is register-based and optimised for running on devices with limited resources[1]. Each VM instance runs in a dedicated UNIX process with an individual user and group identifier. The Dalvik EXecutable (DEX file) is zipped into an Android Package (`apk` file) together with application specific data, and configuration files, e.g. a manifest.

2.2 Security Architecture

The Dalvik VM is also an essential part of the security enforcement in Android. It centrally monitors security relevant activities such as the control and forwarding of intents, the access to protected APIs, the use of content providers, etc.

Applications use the manifest to specify the permissions required to run. It can request the permission to use specific API functions, forbid other applications to access activities defined by itself, define which broadcast events an application processes, etc. The respective permissions are granted at installation time. This is either done automatically by the system or requires user interaction if the permission to be granted belong to a specific protection level. All permissions must be granted, otherwise the application is not installed. During execution, if an application requested more permissions than specified in the manifest, the Android runtime environment informs the user and terminates the application execution.

[1] Available at `http://www.dalvikvm.com/` (January 2012)

For data storage, every application possesses a separate data directory. Android uses the access control mechanisms provided by Linux to protect the data contained in this directory. Various APIs can be used to store data in files or databases. Data exchange between Android applications is accomplished by content providers or by intents. As mentioned above, access to content providers and intents is also defined in the application manifest.

2.3 Security Issues and Deficiencies

Additionally, applications can define specific access control mechanisms by delegating access to resources. An intent equipped with the correct permissions can be sent to a delegated application to grant access to a resource which is specified by a URI. Delegation is required to temporarily allow access to specific resources without granting complete access to all other resources. This security enforcement works if each developer implements his own security framework performing the correct access decisions. However, individual developers must decide at design and implementation time which security policies would fit the user and application. This generates strong dependencies among applications as applications requesting data always depend on other applications which enforce specific security policies. However, address or media databases, etc. often must offer direct access to stand-alone applications. Therefore, to not restrict the usability of his application, a developer would have to implement a multitude of access control functions or content providers. Only this will guarantee that current and future applications will be able to gain access to the appropriate data.

Thus, the process of assigning permissions in Android is not suitable. The user must decide which permissions an application should possess. From a data-centric point of view, a user has to consider which permissions assigned to an application may compromise his data. Even if the user had the ability to keep an overview of all applications installed in the system, their interaction mechanisms, and if he were aware of the security implications of all permissions, he would not be able to modify the requested permissions. Not accepting or revoking permissions would not prevent the installation of an application or require its de-installation. This coarse grained permission system results in an *all or nothing approach*. Although an application should only have access to a small fraction of data stored in a resource, it either gains access to all information or the application cannot be executed.

The situation is even worse. Permissions granted by the user or system implicitly generate trust chains between applications. In this way, applications can gain access to data items although they did not explicitly ask for it. As a consequence, the execution of an application may directly conflict with the security requirements a user implicitly expects for specific data items.

2.4 TaintDroid

TaintDroid is a taint tracking extension for Android optimized for Smartphones performance. It is mainly enabled by modifying the Dalvik VM and introducing taint tracking on the variable level. For this purpose, TaintDroid introduces shadow variables. As the VM stores local variables and arguments on an internal stack, the original taint tracking system modified this stack structure by doubling the space each variable requires on the stack. This additional space is used for a 32-bit taint value.

The 32-bits of the taint values are used to identify the privacy critical data sources which influence a data item, e.g. the contact provider or data in the telephony or location manager. To set the appropriate taint bits, TaintDroid modifies the system libraries which retrieve the requested data. These modifications modify the appropriate taint values in the Dalvik VM. These taint values are processed in a similar way at the data sinks, e.g. network sockets. Before a variable is processed by a data sink the respective taint value is retrieved. Its value is written to the system log and an appropriate warning is generated if privacy related information leaks.

Kynoid uses the infrastructure provided by TaintDroid, i.e. we use the modifications within the Dalvik VM and adjust or extend them for our purposes. The next sections describe these adaptations.

3 Approach Overview

Our work is based on the tracking framework TaintDroid [5]. TaintDroid tags data sources in Android, e.g. the address book, or the browser data, and tracks them during runtime. These tracking mechanisms are enabled by modifications of the underlying Dalvik virtual machine and the corresponding libraries which form the interface between the applications and the execution environment. The tracking capabilities can be used to log applications which gain access to the data sources of Android and misuse the information stored therein. However, this approach, even if it was used for permission enforcement, such as in AppFence [9], is far too coarse grained to effectively support a practical permission system which allows for resources containing security critical and non-critical data at the same time.

Recollect our motivating example in Section 1. A shared resource, such as the address book, contains private as well as business data. The Smartphone is open for private use and allows the installation of applications which may use this shared resource. Once granted in the Android permission system, an application may simply use all the information stored in this resource. Thus, all data items stored in this resource share the same permission. Similarly, TaintDroid and AppFence, use a one-fits-all paradigm to track and enforce the information usage, e.g. to prevent its transmission to some recipients. However, this is not feasible for real life scenarios. In fact, different data items stored within a shared resource often have different security requirements. Therefore, Kynoid extends TaintDroid to support the tracking of security policies for single data items.

TaintDroid uses an approach which is comparable to shadow variables (see Section 2.4). They only exist in the interpreter and store the taint tags associated with a variable. Special instructions which are located at the data sources set particular bits in these taint tags. Every bit accounts for a particular data source. During the run of the application, simple bit operations combine the bit fields depending on the type of instruction executed in the interpreter. At the sink, the single bits of the taint tags are evaluated and reported. In this approach, the number of different taint tags is limited to the number of bits in the shadow variable. TaintDroid currently supports a 32 bit field and thus 32 different tags, which can only cover the main data sources in Android.

We decided to use a more generic approach and use the 32 bit space of the shadow variable for identifiers (ID). In this way, we associate each variable in an Android executable with an ID which is again associated with a policy. This allows us to basically map 2^{32} variables to an arbitrary number of security policies (depending on how security policies are identified).

However, this simple change in the interpretation of the shadow variable yields some problems. The policy propagation becomes slightly more complicated in comparison to TaintDroid. Assume an interpreter instruction in the dex file, such as an addition, on two variables which carry two different taint tags in their shadow variables. The taint propagation logic in TaintDroid simply combines the shadow variables by using a bitwise `OR`. This sets the bits, which correspond to the appropriate data sources. Thus, the shadow variable of the result of the binary operation now indicates that the result has been influenced by both data sources.

Kynoid does not track simple taints but IDs which indirectly correspond to combinations of security policies. Thus, a simple binary operation on these IDs is not feasible. The efficiency also forbids the evaluation of the corresponding policies with their modal constraints into a new security policy during runtime. In fact, this would also induce tremendous runtime and memory overhead as the generation and registration of an instance of a data structure which stores this security policy would also be required. As a consequence, Kynoid builds a dependency graph which uses a simple graph structure to store the runtime combinations of policies. At the sink the appropriate graph is evaluated to derive the correct security policy.

4 Kynoid

Kynoid is the prototypical realisation of our dynamic, fine-grained policy tracking system. The next sections explain how this *watchdog* administrates, retrieves, tracks, and enforces security policies for data processed by an untrusted application. We introduce the Kynoid architecture and its modifications.

4.1 Framework Operation

Figure 1 gives an overview on the high-level architecture of Kynoid and sketches the modifications of the Android runtime environment to allow for the tracking of security policies. We explain the single entities of this architecture and their interaction by assuming the execution of an application. Although Google just

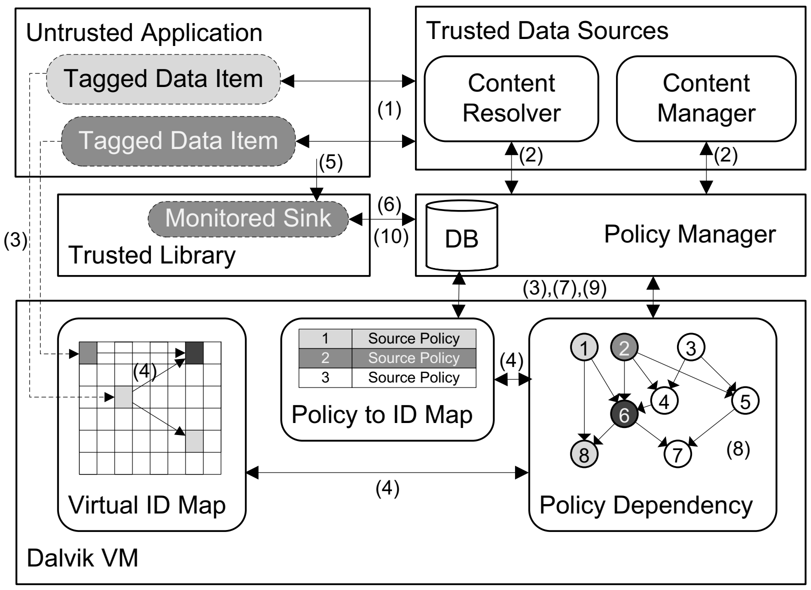


Fig. 1. Kynoid Architecture

introduced new mechanisms which allow the automatic verification of applications, we assume Android applications to be untrusted. This is particularly true for applications which stem from third-party-markets. Thus, the installed application can contain potentially malicious execution paths. It can use shared data sources (1) by using so called content providers or by accessing content managers. Content managers and content resolvers, which content providers connect to, are considered to be trusted data sources, i.e. they are delivered with the Android OS. As soon as these trusted components access data, they also query the Policy Manager (2) for a reference to the security policy associated with the requested data items. The policy manager then interacts with the Dalvik VM (3) and registers this policy reference with a unique identifier. It corresponds to the variable which contains the data item requested by the untrusted application. This association is stored in the policy to ID map.

During execution, Kynoid must correctly propagate (4) the identifiers which link the variables to the corresponding policies. For this purpose Kynoid uses a Policy Dependency graph. It stores binary dependencies among policy IDs. Every node in the graph represents a variable. A directed edge from node n to node k denotes that the information of the variable represented by node n flows into node k. Thus, the security requirement of node k depends on the security requirements for node n. More precisely, the security requirements which hold for node n must also hold in node k. Nodes without incoming edges are source nodes, i.e. their security policy is explicitly defined by the security constraints stored in the policy manager. Such policies are called *source policies.*

If an interpreter instruction or sequence of instructions combines two variables we would have to compute a new policy which holds for this variable. The computation of this policy during runtime would induce tremendous overhead as the Dalvik VM would either have to query the security policy for each item from the policy manager and administrate all newly generated policies. Using the dependency graph, such computations boil down to the creation of new nodes and their corresponding edges in the graph. Compared to the computation of a complete policy, the runtime overhead is negligible. However, depending on the execution flow and runtime of an application the memory overhead may become substantial. Therefore, Kynoid only builds dependency graphs for variables which contain information influenced by data-items which carry security policies.

Also, the complete administration of the policies by the Dalvik VM would not be feasible as the policy may change during execution. Thus, the security requirements loaded in the Dalvik VM may already be obsolete. Therefore, Kynoid postpones this computation until a data item is actually processed at a sink (5). At this point, Kynoid starts to the derive the security policy linked with this variables. For this purpose, Kynoid can assume all libraries to be trusted which are delivered with the Android system and provide writing access to resources. They basically act as policy enforcmenet points. In our prototype this is the Apache library which allows Android to send `GET` or `POST` requests to remote servers. In general, we could use any type of sink, i.e. other network sinks, e.g. sockets, can be monitored as well as other data sinks such as data base sinks, file system sinks, etc. Thus, as soon as the application tries to send a request the trusted library contacts the policy manager. The latter resolves the ID (7) delivered by the library and uses the dependency graph (8) to infer the correct policy for the processed variable. For this purpose we access the node in the graph which represents this variable and determine all of its predecessors. As soon as we arrive at source nodes a list of policy references associated with these nodes can be returned to the policy manager (9). The policy manager can now compute the correct security policy and enforce it by joining all these policies.

As an example consider node 6 of the dependency graph in Figure 1. To derive the security constraints for the variable associated with this node the source policies associated with the nodes 1,2, and 3 are required.

4.2 Policy Propagation

To correctly propagate the policy identifiers, Kynoid uses a concept which is comparable to the variable-level taint tracking within the Dalvik VM applied in TaintDroid. It mainly distinguishes in the values propagated during execution. Instead of tracking bit fields, which represent taint values, Kynoid tracks identifiers. As mentioned above, these identifiers can be associated with policies or their combinations. Thus, we have to introduce an appropriate propagation logic. This data flow logic does currently not consider implicit flows.

We assume $\Im$ to be the set of possible policy identifiers for our system. For efficiency reasons, the virtual memory address of a variable represents the identifier for this variable. This has the advantage that we do not have to ensure uniqueness of the identifiers within one process and several access operationscan be performed in $O(1)$. As mentioned in Section 2.4, the Dalvik VM offers five variable types. We choose the same representation as in [5] and denote local and argument variables by v_x. In the virtual machine, they represent virtual registers. Further, field variables of a class x are denoted as f_x. Without the indication of an instance object, f_x denotes a static field. If v_x denotes an instance object, $v_x(f_x)$ denotes an instance field of object referenced in v_x. Finally, $v[\cdot]$ denotes an array, where the variable v_x contains the reference to this array.

To access the policy identifiers of the variable types listed above we define the virtual identifier function ι. $\iota(v)$ returns the identifier associated with variable v. At the same time $\iota(v)$ can also assign a policy identifier to variable v. The interpretation of ι depends on its position in respect to the $\leftarrow$ symbol. Located on the right hand side of $\leftarrow$, $\iota(v)$ reads the identifier for variable v. On the left hand side $\iota(v)$ is used to set the identifier value for v. We further define the graph $G = (E, V)$ with the node set $V \subseteq \Im$ and the directed edges $E \subseteq V \times V$. It denotes the directed dependency graph used for storing the policy dependencies. To update this graph we define the function $\theta : \wp(G) \times \Im \leftarrow \wp(G) \times \Im \times \Im$ in the following way: $\theta(G', c) \leftarrow (G, a, b)$ with $a, b, c \in \Im, G' = (V \cup \{c\}, E \cup \{(a, c), (b, c)\})$. Here, $\wp(G)$ denotes all valid dependency graphs. With this definition θ creates a new node in the dependency graph and two direct edges which start at the already existing nodes a and b and end in node c and returns the modified graph and the new node, i.e. the policy identifier. Thus, $(G, \iota(v_c)) \leftarrow \theta(G, \iota(v_A), \iota(v_B))$ will modify the dependency graph by inserting the new node $\iota(v_c)$ and adding the edges $(\iota(v_A), \iota(v_C))$ and $(\iota(v_B), \iota(v_C))$.

With this information we can read the propagation logic in Table 1. To reduce redundancy and remain readable we only list the abstracted byte code versions of the instructions described in the DEX specification. Our propagation logic is very similar to the one presented in [5]. This is due to the fact that regular taint propagation is identical to policy identifier propagation if the content of the respective markings is not altered. However, as soon as there are combinations of policy identifiers, i.e. for any binary, array, or field operation, we have to modify the logic.

4.3 Inter-process Policy Tracking

Inter-applications communication is common in Android. Therefore, TaintDroid also propagates taint tags between processes (each VM runs in a separate process). To maintain overall efficiency, TaintDroid uses message level tainting, i.e. a message is assigned the upper bound of all taint tags of the variables contained in the parcel. Of course, this overestimation can generate false positives during execution. We aim at a more precise tracking system which stays on the variable level and does not decline to message-level tracking. To remain efficient, Kynoid uses a simple inter-policy-process mapping (see Figure 2). It simply maps the policy identifiers of the variables exchange between the two processes. As the policy identifiers correspond to the memory address of the variables, this mapping

Table 1. Policy Propagation Logic. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter. C is a byte-code constant.

Op Format	**Op Semantics**	**Taint Propagation**	**Description**
const-op $v_A C$	$v_A \leftarrow C$	$\iota(v_A) \leftarrow 0$	Delete v_A ID
move-op $v_A v_B$	$v_A \leftarrow v_B$	$\iota(v_A) \leftarrow \iota(v_B)$	Set v_A ID to the ID of v_B
return-op v_A	$R \leftarrow v_A$	$\iota(R) \leftarrow \iota(v_A)$	Set the ID for the return value to v_A (0 if void)
move-op-R v_A	$v_A \leftarrow R$	$\iota(v_A) \leftarrow \iota(R)$	Set v_A ID to the ID of the return value
throw-op v_A	$E \leftarrow v_A$	$\iota(E) \leftarrow \iota(v_A)$	Set the ID of the exception taint to v_A
move-op-E v_A	$v_A \leftarrow E$	$\iota(v_A) \leftarrow \iota(E)$	Set v_A ID to the ID of the exception value
unary-op $v_A v_B$	$v_A \leftarrow \otimes v_B$	$\iota(v_A) \leftarrow \iota(v_B)$	Set v_A ID to the ID of v_B
bin-op $v_A v_B v_C$	$v_A \leftarrow v_B \otimes v_C$	$(G, \iota(v_A)) \leftarrow \theta(G, \iota(v_A), \iota(v_B))$	Create new ID for v_A and store dependency on $\iota(v_A)$ and $\iota(v_B)$
bin-op $v_A v_B C$	$v_A \leftarrow v_B \otimes C$	$\iota(v_A) \leftarrow \iota(v_B)$	Set v_A ID to the ID of v_B
aput-op $v_A v_B v_C$	$v_B[v_C] \leftarrow v_A$	$(G, \iota(v_B[\cdot])) \leftarrow \theta(G, \iota(v_B[\cdot]), \iota(v_A))$	Create new ID for $v_B[\cdot]$ and store dependency on $\iota(v_B[\cdot])$ and $\iota(v_A)$
aget-op $v_A v_B v_C$	$v_A \leftarrow v_B[v_C]$	$(G, \iota(v_A)) \leftarrow \theta(G, \iota(v_B[\cdot]), \iota(v_C))$	Create new ID for v_A and store dependency on $\iota(v_B[\cdot])$ and $\iota(v_C)$
sput-op $v_A f_B$	$f_B \leftarrow v_A$	$\iota(f_B) \leftarrow \iota(v_A)$	Set field ID for f_B to ID of v_A
sget-op $v_A f_B$	$v_A \leftarrow f_B$	$\iota(v_A) \leftarrow \iota(f_B)$	Set ID of v_A to ID of field $\iota(f_B)$
iput-op $v_A v_B f_C$	$v_B(f_C) \leftarrow v_A$	$\iota(v_B(f_C)) \leftarrow \iota(v_A)$	Set field ID for f_C to ID of v_A)
iget-op $v_A v_B f_C$	$v_A \leftarrow v_B(f_C)$	$(G, \iota(v_A)) \leftarrow \theta(G, \iota(v_B(f_C), \iota(v_B))$	Create new ID for v_A and store dependency on $\iota(v_B(f_C))$ and $\iota(v_B)$

process is very efficient. To resolve the security policy at the sink and allow for its evaluation Kynoid simply queries the corresponding processes to determine the identifiers of the involved source policies. The policy manager can resolve those and enforce the security policies accordingly.

Apart from this, the IPC in Kynoid communication is similar to TaintDroids' IPC. It modifies the binder library which is used by Android to accomplish inter process communication. A hook takes care of the modifications in the message exchanged between the processes and the modifications of the policy-process mapping.

4.4 Policy Derivation and Enforcement

So far we did not present the security policies defined in Kynoid. This shows another strength of our tracking system. In general, it is independent of the type of security policy to be enforced. For completeness, we consider a simple policy specification language which can describe policies with temporal, spatial, and remote host constraints. They are defined on a data-item with identifier i and consist of triples of the form $(T, S, H)_i$ where T is a set of time intervals, S is a set of circular perimeters, and H is a set of IPs of remote hosts. If a set is empty, no constraint is defined for the respective modal dimension. If a set contains more than one element, the elements are interpreted as disjunctions. We further define the operator $\wedge$ on the policies which is similar to the interpretation known from Boolean logic. $(T, S, H)_i \wedge (T', S', H')_k$ means that all constraints specified for data-items with identifiers i and k have to hold.

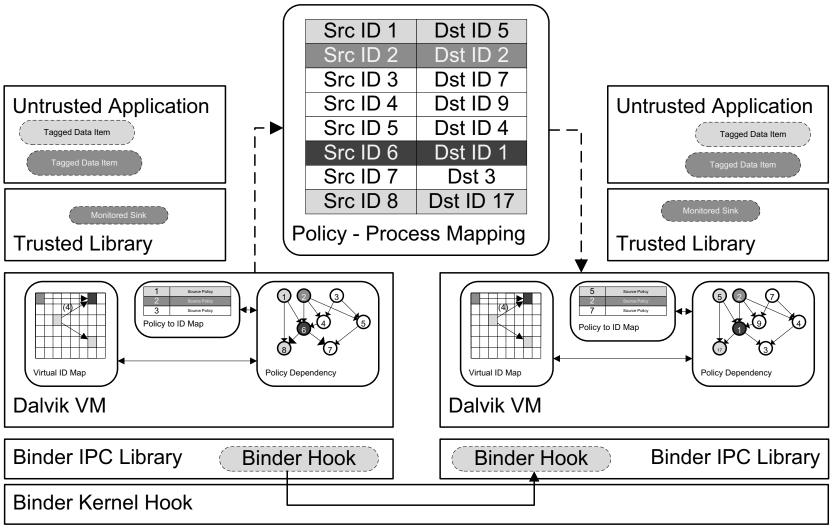


Fig. 2. Inter-Process Policy Tracking

As already sketched in Section 4.1 we use the dependency graph $G = (V, E)$ to determine the policies which have to be evaluated if a variable v is processed at the sink. For this purpose we define the function $Pred : \wp(G) \times \Im \rightarrow \wp(\Im)$ which delivers all predecessor nodes with no incoming edges, i.e. those variables for which a security policy has been defined in the policy manager. It can be used to determine the set P of variables which influenced v: $P = Pred(G, i_v)$. To determine whether variable v can be used in the execution context, the policy manager only has to verify that the policy $\bigwedge_{p \in P}(T, S, H)_p$ holds.

While our concept can be adapted to different types of sinks, we currently only consider network sinks. Thus, as soon as an application tries to send information stored in variable v using the network interface the policy $(T, S, H)_v$ is informally interpreted as: *Only send v to one of the hosts with IP addresses listed in H if the current time is in one of the time intervals specified in T, and if the phone is located at one of the locations specified in S*. In this case, the default action, if the policy does not hold, is to reject the further processing of the data item v. In fact, our proof-of-concept implementation also supports policies which have a positive default action, i.e. the further processing is only allowed if all constraints do not match. However, to maintain simplicity and due to spatial constraints we do not define these rules here.

5 Discussion

This section discusses Kynoid in terms of its added value to the existing Android platform, its performance, and in terms of its usability.

5.1 Applicability

We experimented with well known applications which tend to synchronise user data, e.g. the Facebook App. If the user does not pay attention during its first run this application may synchronise the complete address book with the facebook servers. Kynoid allows to select the contacts we like to synchronise with Facebook. Using Kynoid's administration interface, we can select our business contacts and specify that they should only be transmitted to a specific remote host. If we now start our Facebook application again it is still able to synchronise the data stored in our contacts database. However, Kynoid blocks all connections if they contain data items we marked to be business contacts. This simple example shows that even with the rudimental user interface we offer in our prototype, it is already possible to provide an added value in comparison to existing solutions.

5.2 Performance

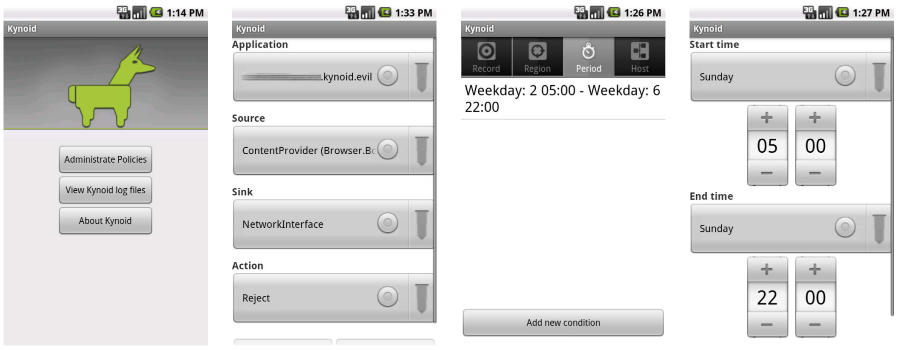
When we started this work, the main concern was the runtime overhead induced by the tracking of complex security policies. Kynoid efficiently addresses this problem by using a graph structure which stores the flow of information between variables. It avoids policy evaluations every time more than one variable which is associated with a security policy writes into another variable. Instead of evaluating the policies during policy propagation, Kynoid only evaluates policies where they are enforced. This is done by a fast graph resolution which determines the source policies which influence a graph node. To measure this overhead we implemented a benchmark application which reads 100 entries from the bookmark database. This generates a basic dependency graph in the Dalvik VM. This benchmark was executed 50 times on a Nexus One with Android 2.2 patched with Kynoid. This delivered an average runtime of 2376 ms with a standard deviation of 89 ms. The same benchmark, was also run for our Kynoid proof-of-concept implementation with disabled tracking capabilities, i.e. no dependency graph was created. This generated an average runtime of 2292 ms with a standard deviation of 91 ms. Thus, the generation of the dependency graph for the policy IDs produces an overhead of 3.7%.

The runtime overhead produced by the propagation of policy identifiers is negligible in comparison to the TaintDroid implementation. The benchmarks applied to our proof-of-concept did not produce meaningful results in this respect. This is not surprising as the taint propagation is very similar to TaintDroid (see also Section 4.2). The memory overhead generated by the dependency graph strongly depends on the implementation of an application. An appropriate study of different applications executed by our framework can deliver appropriate estimations of the average memory overhead.

Major overhead is generated by the data base queries of the policy manager. On an Android 2.2 stock image our benchmark shows an average runtime of 740 ms with a standard deviation of 52 ms. Thus, our proof-of-concept implementation of Kynoid produces an overhead of 321% with the selected benchmark. This is also not surprising as the non-optimised policy manager induces tremendous overhead as it administrates the security policies in a database. In turn, database access is performance critical in Android. Although, the policy management was not focus of this work, we are positive to be able to reduce this overhead. We have already shown such a performance optimization for Constroid [19]. In this security policy management system, we were able to reduce the runtime overhead to $21 - 22\%$.

5.3 Usability

Most of the operations of Kynoid are transparent. In particular, the policy propagation is completely invisible for the user. Also the enforcement is mainly transparent although our prototype generates some warning messages to inform the user about blocked requests not compliant with the defined security policies.



Application
[redacted].kynoid.evil
Source
ContentProvider (Browser.Bo
Sink
NetworkInterface
Action
Reject

Weekday: 2 05:00 - Weekday: 6 22:00

Start time
Sunday
05 00
End time
Sunday
22 00

(a) Dashboard (b) Policy Creation (c) Constraint view for the security policy of a single data item (d) Specification of a temporal constraint

Fig. 3. Kynoid administration tool

However, for the front-end, the Kynoid proof-of-concept implementation provides a graphical user interface. Figure 3 shows some screenshots of the basic functionalities of the front-end. Through a dashboard (see Figure 3(a)) the user can administrate the security policies specified for the data-items of different data sources. To create new policies the user must first select (see Figure 3(b) the source for the data items for which policy constraints should be defined. After also selecting the type of sink for which the policy should be created in a specific application, the user can also determine the action to be performed when the policy matches. In Figure 3(b) we will define a policy for a data item in the browser database, and define a rejection policy for the network sink. On the next screen, Figure 3(c), the user can use the different tabs to define the various temporal, spatial, and host constraints for the data item he chose on the record tab. Exemplarily, Figure 3(d) show how the user can define a temporal constraint.

6 Related Work

The body of literature in the realm of taint tracking for policy enforcement is huge. Due to spatial constraints, this section focuses on the most relevant approaches.

Panorama [21], Trishul [12], TaintDroid [5], and AppFence [9] are very similar to Kynoid. This is not surprising as Kynoid is based on TaintDroid. All of these frameworks track security properties. However, the granularity of all these approaches stay on the process level and or a coarse grained resource which does not allow for much flexibility.

Porscha [15] and T-UCON [13], 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. Secondly, Porscha and T-UCON only mediate the exchange of data between applications, Kynoid ensures compliance of data usage during execution. The European project S3MS also generated a series of publications [2,3,4,17]. They implement the concept of security by contract, specifying policies according to which an application should behave. Of course, this approach is application-centric as well.

Saint [16] and Apex [14] extend Androids permission mechanism by allowing to define rules for granting permissions and by allowing the denial of permission subsets. However, these constraints are static, enforced at runtime, and maintain the coarse granularity of the Android permission system. The same holds for Kirin [6]. It is a security service which analyses the configuration of an Android application and detects potential security issues. It does not track information to enforce security. Finally, Constroid [19] also defines a management framework for data-centric security policies of fine granularity. However, it also does not show how to track and enforce these policies.

Other platforms such as Asbestos [20], HiStar [22], Flume [10] perform information flow tracking in decentralised environments, however their architectural overhead is large compared to their granularity which also stay on the process and resource level.

Hence, Kynoid clearly distinguishes from related work by deploying data-centric and user-defined policies which can be tracked and enforced efficiently.

7 Conclusions and Future Work

The tracking of security related information in Android was limited to taint tags. For this purpose, TaintDroid provided an efficient architecture. However, TaintDroid is limited to a set of at most 32 different data sources. AppFence was the first approach based on TaintDroid which tried to abolish this coarse granularity. However, also AppFence sticks with the coarse granularity already known from TaintDroid. Kynoid, breaks with these pure taint tracking approaches and introduces finer granularity. It also defines security policies on the variable level and is the first general approach to show that the dynamic tracking of security policies of this granularity is feasible.

Although Kynoid currently only exists as a prototype implementation we can already show its potential impact on the privacy concerns of users. Smartphone owners can define security policies for single data items and specify their appropriate use. If an application does not stick to these security constraints, Kynoid can transparently block respective actions. Future implementations may also query the user for specific actions. These actions may also be used to adapt the policies of the respective items.

We also intend to couple our system with the capabilities, performance, and security characteristics of Constroid [19]. It provides a middle-ware layer which could be used to efficiently and securely retrieve the security policies required for Kynoid. The power of Kynoid will further be increased by supporting a larger number of API sources and sinks.

Finally, future work will also investigate the impact of indirect flows to the overall performance and precision of the presented policy tracking system.

Acknowledgements. We would like to thank the students Tobias Marktscheffel, Michael Klassen, Fabian Kokot, and Steffen Kremer for their excelent work in helping to implement this framework. Without their support, we would not have been able to provide this contribution.

References

1. Apple Inc.: Security Overview. Tech. rep., Cupertino, CA, USA (2010)
2. Castrucci, A., Martinelli, F., Mori, P., Roperti, F.: Enhancing Java ME Security Support with Resource Usage Monitoring. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 256–266. Springer, Heidelberg (2008)
3. Costa, G., Lazouski, A., Dragoni, N., Saadi, R., Ingegneria, D.: Security-by-Contract-with-Trust for Mobile Devices. Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications (JoWUA) 1, 75–91 (2010)
4. Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., Vanoverberghe, D.: Security-by-contract on the.NET platform. Information Security Technical Report 13(1), 25–32 (2008)
5. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proceedings of OSDI 2010, pp. 1–6. USENIX Association, Vancouver (2010), `http://appanalysis.org/tdroid10.pdf`
6. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 235–245. ACM Press, New York (2009)
7. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security & Privacy Magazine 7(1), 50–57 (2009)
8. Heath, C.: Symbian OS Platform Security, Software Development Using the Symbian OS Security Architecture. John Wiley & Sons Ltd. (2006)
9. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 639–652. ACM, New York (2011)

10. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information Flow Control for Standard OS Abstractions. In: Proceedings of ACM Symposium on Operating Systems Principles (2007)
11. Microsoft Corporation: Windows Phone 7 Security Model. Tech. rep. (December 2010)
12. Nair, S., Simpson, P., Crispo, B., Tanenbaum, A.: Trishul: A Policy Enforcement Architecture for Java Virtual Machines. Tech. rep., Vrije Universiteit, Amsterdam, Netherlands (2008)
13. Nair, S., Tanenbaum, A., Gheorghe, G., Crispo, B.: Enforcing DRM policies across applications. In: Proceedings of the 8th ACM Workshop on Digital Rights Management - DRM 2008, p. 87. ACM Press, New York (2008)
14. Nauman, M., Khan, S., Zhang, X.: 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, pp. 328–332. ACM Press, Beijing (2010)
15. Ongtang, M., Butler, K., McDaniel, P.: Porscha: Policy Oriented Secure Content Handling in Android. In: Proceedings of the 26th Annual Computer Security Applications Conference. ACM Press, New York (2010)
16. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. In: 2009 Annual Computer Security Applications Conference, pp. 340–349. IEEE Computer Society (2009)
17. Philippaerts, P.: Security of Software on Mobile Devices. PhD thesis, Department of Computer Science, Faculty of Engineering, Leuven, Belgium (2010)
18. 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 (2009)
19. Schreckling, D., Posegga, J., Hausknecht, D.: Constroid: Data-Centric Access Control for Android. In: Proceedings of the 27th Symposium on Applied Computing (SAC): Computer Security Track (2012)
20. Vandebogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., Mazières, D.: Labels and Event Processes in the Asbestos Operating System. ACM Transactions on Computer Systems (TOCS) 25 (2007)
21. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 116–127. ACM Press, New York (2007)
22. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making Information Flow Explicit in HiStar. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI (2006)