Security Policy Synthesis in Mobile Systems

Peter Amthor  
Ilmenau University of Technology  
Ilmenau, Germany  
peter.amthor@tu-ilmenau.de

Winfried E. Kühnhauser  
Ilmenau University of Technology  
Ilmenau, Germany  
winfried.kuehnhauser@tu-ilmenau.de

Abstract—Contemporary mobile devices have become universal and versatile tools that increasingly are used in sensitive application scenarios. They inevitably carry confidential information such as passwords, encryption keys, mission-critical company data, or location information in combat areas. In order to meet sophisticated security requirements, recent technology focuses on policy-oriented approaches that allow for the definition and enforcement of rigorous and precise rules for protecting confidential information.

State-of-the-art development of security policies is a critical process; because of the involved quality assurance measures, it is quite heavy-weighted and tends to antagonize the distinguished virtues of mobile devices for lightweight, spontaneous communication and cooperation. This paper presents an approach to support secure, mobile device based cooperation in temporary, sporadically and spontaneously fashioned cliques within open communication infrastructures. The approach is based upon light-weight security domains protected by security policies that are dynamically and automatically composed during group formation. Due to the volatile nature of such groups simplicity, adaptability, efficiency and compatibility with today’s security policy implementation techniques have been a major design goal.

Keywords-Mobile computing; security policy; security domain; metapolicy; mobile devices security; ad-hoc cooperation; policy-controlled system; SELinux; SEAndroid; Android Security Extensions; Flaskdroid; MOSES

I. INTRODUCTION

In the last decade, mobile devices have become sufficiently powerful to frequently be used in application scenarios that involve sensitive information. In the private sector for example, they are used for storing contact information, passwords for email accounts or electronic banking, or airport check-in information; in government, enterprise, or military scenarios the privacy, integrity and availability of information carried by mobile devices is often mission-critical.

For more than a decade now, IT systems with advanced security requirements apply problem-specific security policies for specifying, analyzing, and enforcing strategic security concepts, and paradigms have emerged that are capable of enforcing security policies directly by operating systems [1], [2], [3]. Given the critical importance of policy correctness in such systems, policy engineering approaches based on formal methods and formal security models have established [4], [5], [6], [7], [8], that allow for a verified specification and implementation of security mechanisms and architectures. Currently, this methodological and technological foundation is adopted to enforce rigorous security properties in mobile platforms [9], [10], [11], [12], [13], [14].

However, mobile platforms throw a new light on policy-based protection paradigms, suggesting that current policy-supporting approaches are not always feasible. Different operational conditions (distributed systems including mobile components) and different applications (e.g. considering environmental conditions such as geographical location) result in additional threats and vulnerabilities. Mobile devices are generally not physically protected and can be stolen easily; they command a multitude and diversity of communication alternatives using untrusted or even hostile communication infrastructures. Communication is often spontaneous, volatile and unpredictable, and communicating parties frequently are only weakly authenticated or downright hostile.

Current policy-based approaches have a major drawback under unpredictable operational conditions. Due to their key role in protecting information, security policies are designed and implemented carefully, by applying formal models and rigorous implementations based on reference monitors. The engineering process as well as the implementation principles cast a policy in steel: changing a system’s security policy is expensive and time-consuming, which is quite an obstacle for the fast-changing, highly dynamic operational conditions of mobile systems. On the other hand, anticipating the multitude of future situations in the policy design phase is just not feasible, firstly because of the resulting increase in policy complexity and secondly because completely foreseeing the future is not possible. Consequently, policies have no change management.

This paper’s concern is a sub-problem of the dynamic and unpredictable operational conditions of mobile devices that is typical for many mobile applications. Airport check-in, electronic payment, or car-to-X communications all involve a sporadic, spontaneous and volatile fashioning of cooperating groups in potentially hostile environments. For such scenarios the traditional way of specifying, implementing and verifying anticipatory security policies for each such cooperation is unfeasible; spontaneity implies that there will be scant prior warning and no sufficient a priori knowledge.

This paper presents an approach to support the fashioning
of sporadic and spontaneous cooperation groups by lightweight security domains controlled by security policies that are automatically generated on demand. The overall idea is that whenever a group is formed a rule set is applied that creates an individual short-term security domain for the group and assembles a security policy governing any interaction in that domain. The contributions in this paper are twofold. Firstly, the approach takes up work on metapolicies and policy groups [15], [16], [17] and forges it into a method to dynamically compose security policies for spontaneously fashioned and volatile cooperating groups. The approach uses precise algebraic specifications for describing composition rules, allowing for spontaneous, automated and transparent policy composition whenever a cooperation group is formed. Secondly, the feasibility of the approach is demonstrated by a proof-of-concept implementation on a MOSES platform [10] using contemporary state-of-the-art technology.

II. APPLICATION SCENARIOS

This section sketches two scenarios that illustrate security risks associated to spontaneously formed groups. Excerpts of these scenarios are referenced throughout the paper.

Consider a manager of a large insurance corporation who plans and initiates strategic alliances with business partners. To this end she frequently meets with representatives of candidate companies. On her smartphone she carries (1) a presentation prepared for these meetings that she will send to an LCD projector in the meeting room via her host’s communication infrastructure, (2) sensitive charts detailing financial data of her own company that she will only reveal gradually during a successful course of the meeting.

In her home office the manager’s smartphone is an integrated part of her company’s IT infrastructure. Any information on the device is part of her company’s security domain and thus subject to a corporate security policy. During the meeting however, the smartphone is exposed to the infrastructure of the hosting company.

The risks involved in this scenario are twofold. On the one hand there are risks for the host’s IT systems: the visitor’s smartphone now is within firewalls that normally protect the hosting company’s security domain. The smartphone might exploit this situation for attacking the local systems through spying agents, or by implanting Trojan horses or backdoors. On the other hand, the smartphone encounters similar risks, because key components of its home domain’s security infrastructure (authentication servers, authorization servers, firewalls) are missing, the communication infrastructure is controlled by a potentially malicious party.

Other scenarios of spontaneous cooperation arise by the emergence of collaborative media sharing and delivery to mobile devices. One example is the approach of decentralized Device-to-Device content delivery to address increasing load on mobile network infrastructure due to bandwidth-consuming multimedia data. The idea is to use decentralized, short-range device communication to deliver common media content on locally near end-user smartphones. To reduce traffic on mobile networks, content is downloaded to one nearby smartphone and then distributed to other devices in range in a self-organized manner [18], [19]. Since this basically requires smartphones to deliberately exchange information without user interaction, an implementation must respect security requirements of all involved devices. Besides those risks already discussed above, user privacy with respect to specific content disclosed to other participating devices is of special importance here [20].

Both scenarios illustrate environment-specific operational risks of mobile computing that are well known today. In order to deal with them, state of the art technology uses security policies that regulate access to sensitive information. In our example scenarios, any interaction between two or more devices is subject to at least two different security policies, governing e.g. a smartphone’s security domain and that of an untrusted host. Both policies in general have been developed independently but need to cooperate in a sporadic manner. This may result in conflicting situations; a smartphone for example might be willing to send a presentation file to the host, while the host’s security policy only accepts files that carry a trusted digital signature. Moreover, situations may arise that cannot be dealt with by any of both security policies; as an example, both parties may fail to understand or trust the other’s security labels and thus cannot apply any policy rule.

There is a plethora of specialized implementations for policy-controlled smartphone operating systems, such as MOSES [10], SEAndroid [12], FlashDroid [11], ASF [13], and ANDRABEN [14] just for the Android platform. Moreover, the problem of access control policies that adapt to multiple security domains has been addressed on the level of network access, e.g. based on firewall policy configuration [21]. However, each of these approaches primarily focuses on a single device’s security goals and functionality, rather than those of a dynamically cooperating group. From a methodical point of view, the problem of cooperating security policies is either yet unsolved or has heavy-weight metapolicy solutions that work well only in scenarios that to a large extent are static [22], [23], [16], and where cooperation is neither spontaneous nor sporadic [24], [25], [26]. As a consequence, while there is efficient technology such as mobile devices, mobile objects, infrared and Bluetooth links or wireless LANs the above scenarios are still handled in a rather anachronistic way: by reverting to USB memory, physical connections or low-bandwidth network infrastructure.

Similar limitations can be observed in scenarios where, instead of smartphones, mobile code carrying its own security policy migrates into foreign security domains. Here, the
limits of manual assistance (such as plugging in memory sticks) become even more obvious, since the arrival of mobile code occurs substantially more frequent and without physical interaction or even awareness by human beings.

III. MAPPING POLICIES

This paper discusses an approach for policy-controlled cooperation in spontaneously formed groups. Fundamental to this approach are metapolicies that lodge in the Trusted Computing Base (TCB) of each participating system and tailor the generation of individual security domains and policies for each individual spontaneous cooperation. Generation of a group-specific security policy is done automatically by composing partial security policies that are provided by the individual group members. The composition rules are described by a simple policy algebra whose basic set is the set of involved security policies and whose operations combine elements of this basic set into a new policy. The algebraic approach for specifying metapolicies allows for a fully automatized policy generation carried out efficiently and transparently and without user interaction.

Whenever a group is formed spontaneously, in order to compose the group’s security policy the policy composition rules are applied. A security domain is created that encompasses the group members, and the policy resulting from the ad-hoc composition is associated to the domain. As a simple example, composition rules may describe that for every interaction within a newly formed group, each partial policy provided by the group members must agree.

In a nutshell, the general idea is to design and implement composition rules just once by qualified security personnel and then apply these rules whenever a spontaneous group is set up. The evaluation of these rules then creates a new security policy that is associated to the ad-hoc security domain created for the new group.

This section outlines the policy composition algebra underlying the approach.

A. Policy Composition Algebra

The composition algebra defines the semantics for of policy composition rules. Its components are security policies and operations that allow for a specification of rules that precisely describe how policies provided by the group members are combined into a single cooperation policy. With respect to the class of security policies considered, our approach focuses on access control policies. Access control policies are universal in that they preserve privacy and confidentiality (by controlling read access), integrity (by controlling write access) and availability (by controlling access to resources).

Access control policies define an access control function (ACF) $f$ and a responsibility area, their policy domain, represented as a set $E$ of entities controlled by the policy. The ACF $f_P$ of a policy $P$ is a mapping $f_P : E^n_P \times A_P \rightarrow \{\text{true}, \text{false}\}$ [27] with the semantics that an action $a \in A_P$ among $n$ entities $\in E_P$ is allowed iff $f_P(e^n, a)$ yields true, where $E_P$ is $P$’s domain. $f_P$ thus precisely reflects a policy’s access decisions.

A convenient way to specify and implement a specific ACF is the access control matrix [27], an $n$-dimensional matrix $m : E^n \rightarrow 2^A$ which yields a set of permissions for any given tuple of $n$ entities. $^1$

Contemporary operating systems or database management systems implement ACFs by access control lists (ACLs) or capability lists. Additionally, the dynamic behavior of $P_i$ may be defined by a deterministic automaton in the style of HRU access control models [28]; however, as the dynamic behavior of a security policy is of no consequence to the policy algebra, for the purpose of this paper a policy’s ACF is completely sufficient to describe its behavior.

If we now choose as the basic set in our policy composition algebra the set $P$ of all access control policies and represent each policy $P_i \in P$ by its ACF $f_R : E^n_P \times A_P \rightarrow \{\text{true}, \text{false}\}$, then the composition of policies can be described by operations on the basic set by:

Conjunction: A policy composition that permits an action iff each of the involved policies agrees is constructed by the conjunction operator “$\wedge$” (read: “policy-and”):

\[
\wedge_p : P \times P \rightarrow P \text{ and } \wedge_p(P_i, P_j) \mapsto P_l \text{ where } f_{P_l} = E^n_{P_l} \times A_{P_l} \rightarrow \{\text{true}, \text{false}\} \text{ and } E^n_{P_l} = E^n_{P_i} \times E^n_{P_j}, n_l = n_i + n_j.
\]

Disjunction: A policy composition that permits an action iff at least one of the involved policies agrees is constructed by the disjunction operator “$\vee$” (read: “policy-or”):

\[
\vee_p : P \times P \rightarrow P \text{ and } \vee_p(P_i, P_j) \mapsto P_l \text{ where } f_{P_l} = f_{P_i} \vee f_{P_j}\text{ and its domain being defined as above.}
\]

Negation: An inverse policy is created by a negation operator “$\neg$” (read: “policy-inverse”):

\[
\neg_p : P \rightarrow \neg_p(P_i) \mapsto P_l \text{ where } f_{P_l} = \neg f_{P_i}.
\]

Selection: An if-then-else selection between two policies that depends on the decision of a third policy is achieved by the select operator “$\select$” (read: “policy-select”):

\[
\select_p : P \times P \times P \rightarrow P \text{ and } \select_p(P_i, P_j, P_k) \mapsto P_l \text{ where } f_{P_l} = \text{if } f_{P_i} \text{ then } f_{P_j} \text{ else } f_{P_k}.
\]

$\neg, \wedge$ and $\vee$ are the well-known boolean operators with their obvious meaning. Together with the general properties of commutativity and associativity and the quantifiers “$\exists$” and “$\forall$” we now have a framework for describing the composition of security policies using algebraic expressions. For example, a group policy $P_G$ where each of the involved partial policies $P_i$ has to agree is composed by the expression

\[
P_G = \bigwedge_{i \in G} P_i,
\]

whereas a policy in which it is sufficient for any of the

$^1$Lampson’s original matrix just had two dimensions.
partial policies to agree is composed by the expression

$$P_G = \bigvee_{i \in G} P_i.$$  

Basically, the above list of four basic operators can be extended as and when required; however, because more complex operations can easily be built from these four basic ones, we are satisfied with this baseline.

B. Policy and Policy Domain Composition

The range of cooperation types can be quite broad. Alongside the scenario in Section II, another example is the purchase of securities, involving a purchaser, a broker, and a credit card company, all being equal partners with full autonomy concerning their part of the transaction and the right of veto. Yet another example is a hierarchically structured group where a group leader has the right to enforce her decisions on the subordinates. Still other scenarios such as voting might require majority decisions. Consequently, it is hardly feasible to apply a universal strategy for group policy formation. Rather, the approach in this paper allows each group member to specify an individual strategy for the formation of group policies and their domains.

Whenever a spontaneous cooperation group is established, each prospective party is expected to provide an individual metapolicy containing a specification for the construction of a group policy. Like any security policy, a metapolicy is designed by a party’s company security team and is embedded into the security architecture of a mobile device, and like any security policy, a metapolicy is implemented using a policy specification language (such as XACML [29], see Section IV).

A metapolicy’s semantic now is described by a tuple $(C,P,D)$ consisting of

1. A policy composition expression $C$ specifying the party’s individual policy composition rules using the semantics as described in Section III-A,
2. An access control policy $P$, the security policy sent into the group policy composition,
3. A set $D$ specifying the entities that the party is willing to yield to the group policy’s security domain.

Following the least privilege principle, a policy $P_G$ for a group $G$ with member tuples $(C_i, P_i, D_i)$ now is built by

$$P_G = \bigvee_{i \in G} C_i \text{ with domain } D_G = \bigcup_{i \in G} D_i.$$  

Each $C_i$ is an algebraic term formed according to the composition algebra in Section III-A, the atoms being policies and the operations being conjunctions (such as $\land_P P_i$), disjunctions, negations, selections, or their combinations.

With respect to the implementation, the actual building and enforcement of the policy is performed locally in each group member, the necessary information being obtained by communicating $(C_i, P_i, D_i)$-tuples among the group members in a policy negotiation phase (see Section IV).

The first scenario in Section II now can be planned from the visiting manager’s (“Alice”) point of view by a tuple $(C_{Alice}, P_{Alice}, D_{Alice})$ where

(i) $C_{Alice}$ is the expression $\land_P P_i, i \in G$, denoting that Alice’s composition rule is the conjunction of the policies of all group members,

(ii) $P_{Alice}$ is a simple policy with the access control function

$$f_{P_{Alice}}(\text{thePresentation}, e, a) \mapsto \begin{cases} \text{true} & \text{if } e \in D_G \\ \text{false} & \text{else} \end{cases},$$

granting only her own presentation full access to each entity in $D_G$.

(iii) $D_{Alice}$ is $\{\text{thePresentation}\}$, submitting only Alice’s presentation to the group domain.

The tuple $(C_{Alice}, P_{Alice}, D_{Alice})$ now phrases that a PowerPoint presentation $\text{thePresentation}$ is sent into the domain of the forged cooperation policy (iii). The presentation is granted full access to each entity in the group domain, while no entity may access the presentation (ii), and the group policy constructor (i) ensures full autonomy of $P_{Alice}$ (conjunction operator “$\land$”). By a similar tuple the host can also prepare for meetings with external visitors. By a constructor similar to (i) the host can preserve its full autonomy and complete the scenario by including the projector server in the group domain and granting all entities in the domain the permission to use it.

On the other hand, scenarios with a common mandatory access control policy are built by common identical constructors $C_i = P_{MAC}$. If all group members belong to the same organization, this easily is achieved by corresponding constructor tuples in the TCBs of all participating devices. In scenarios with autonomous group members, the autonomy of each individual is reflected by the fact that each individual explicitly has to declare its acceptance of the MAC policy.

Last but not least, democratic election scenarios in which each decision is based on majority voting are built in two steps. First, a leader election algorithm [30] determines an election supervisor who then implements the voting scheme of the electoral process. With e.g. $f_{P_{leader}}(\{|\{P_i | f_P_i = \text{true}\}| \geq n/2\})$ this policy implements a majority voting scheme that grant access if more than 50% of the voters agree. In the same way, quorum-based schemes can be implemented; e.g. $f_{P_{leader}}(\{|\{P_i | f_P_i = \text{true}\}| = n\})$ results in a scheme that requires full consensus, each voter withholding a right of veto, while $f_{P_{leader}}(\{|\{P_i | f_P_i = \text{true}\}| = 1\})$ results in an imperative group leader semantics analogous to the MAC example above.

IV. IMPLEMENTATION

In this section, we will present the principal implementation of the algebra-based multipolicy enforcement approach. Due
to its widespread use and flexible functionality, we will focus on the Android operating system for smartphones.

After a brief overview covering the basic technologies, details of our prototypical architecture are presented. We then discuss how the basic algebraic elements of a metapolicy are implemented and show how the functional components interact, using a step-by-step example. Finally, an empirical review of the performance impact of our proof-of-concept system concludes this section.

Based on requirements of the target application scenarios, our implementation has the following goals:
1) Enforcing the reference monitor principles [31] for policy-based access control
2) Evaluating algebraic terms for policy-composition based on distributed access control functions
3) Minimizing resource requirements
4) Maximizing user-transparency.

A. Technologies Used

In this section we provide brief information on the basic technologies used to implement our approach: the Android mobile operating system and the MOSES security framework, that enhances Android access control mechanisms.

Since the goal of our implementation is to study the practical feasibility of algebraic security policy synthesis, we will rely on the MOSES framework for Android, developed by Russello, Conti, Crispo, and Fernandes [10]. We have chosen MOSES from a number of well-known and suitable Android-based policy enforcement projects such as SEAndroid [12], FlaskDroid [11], and ANDRABEN [14] because of practical reasons (including good availability, support by the developers and previous experiences with the framework).

The smartphone operating system Android is based on a modified Linux kernel that provides abstractions of the hardware resources (communication devices, persistent storage, IO-devices such as camera, microphone and display, GPS module, etc.). The Android API is provided by another abstraction layer (often called Android middleware), which consists of a Java-based application framework, C-based native libraries and the Android runtime. Its primary purpose is to interpret byte code of applications (usually written in Java); for this, each application is run in an instance of the Dalvik Virtual Machine (DVM). This lightweight virtualization approach ensures that applications are isolated during runtime, since every DVM instance runs as a separate process in its own virtual address space.

Android provides several communication mechanisms for applications based on principal inter-process-communication (IPC) that is offered by the Linux Kernel. From an application point of view, these mechanisms are once again abstracted by the Android middleware in the form of a generalized interface called Content Provider. Using a content provider, any application can explicitly specify an interface for any data within its address space that should be accessible by other applications. In addition, the Middleware provides application-level access to the Kernel API for file system oder network access via a set of so-called Android core libraries (libcore).

The MOSES framework is an implementation of policy-based access control mechanisms for the Android middleware. Its primary goal is to enforce isolation between logical domains, that may coexist on a single smartphone based on different operation scenarios. One common example is the increasing usage of privately owned mobile devices for work (a practice called “bring-your-own-device”), where data as well as applications have to be strictly separated between a “private” and a “work” domain [10]. Within these domains, custom security policies maintained by a company’s security administrator restrict possible access to data and communication with applications in other domains.

In MOSES, domains are implemented as Security Profiles (SP) that basically represent a collection of applications (such as a contacts manager or a text editor), resources (such as contacts or files on persistent storage) and security policies. The latter specify access control rules for the respective domain similar to an ACF.

B. Overall Architecture

This section discusses a prototypical implementation of the distributed policy synthesis on an architectural level, covering the communication model as well as the basic software architecture in the context of the existing MOSES framework.

Since MOSES already implements the notion of logical security domains, it serves as a viable proof-of-concept framework to build our own implementation on. However, because of a slightly different motivation of MOSES, only local security domains together with local access control policies are supported. In order to integrate our distributed policy synthesis approach into this framework, we made a number of modifications that are outlined in the following. For an overview of the adapted architecture, refer to Fig. 1.

1) Communication Model: From a technical point of view, the approach of a distributed metapolicy essentially leads to a distributed system of fully-meshed topology, where all nodes (here: Android devices) exchange messages representing the tuple \((C, P, D)\) via broadcast. For the evaluation of the metapolicy, the originator of a specific request essentially needs to collect votes from all of its peers in order to compose the overall policy decision \(P_G\) (cf. Sec. III-B). We do this by implementing foreign policies through stub objects, that provide a secure communication interface for on-demand policy requests (e.g. a fresh symmetric session key), possibly including a decision caching mechanism. In this way, the actual policies reside in the protected runtime environment of their physical host device,
which also provides any interfaces and resources needed for its local decision making mechanisms.

2) Architectural Model: The implemented architecture strongly relies on the original architecture of MOSES, that has been modified and extended as needed (see Fig. 1). In the following, we will speak of MOSES-inherent components and mechanisms as Tier-1 (T1), while our additional components and mechanisms are called Tier-2 (T2). While T1 access control respects locally implemented mechanisms based on MOSES security policies, T2 access control refers to mechanisms for distributed metapolicy composition and evaluation. In both cases, the implementation of policy enforcement mechanisms (policy enforcement points, PEP) is separated from policy decision mechanisms (policy decision points, PDP).

For the management of distributed domains in addition to local MOSES domains, Tier-2-Security-Profiles (T2SP) are introduced. These are implemented in a newly created T2SP Store component, that basically mirrors the functionality of the T1SP Store: while T1SPs are still used to achieve isolation between local security domains, T2SPs now carry information about the distributed security domains used to enforce metapolicies. The idea implemented here is to assign every involved group node one T2SP that contains information about its composition rules, its access control policy and entities in its security domain. As already explained, policies that are not local to each node (i.e. each policy \( P_i \) on a node \( i \neq j \)) are replaced by a stub-like policy interface object (PIO) in the T2SP. Internally, MOSES classes for database management and policy access have been modified for this, adding new light-weight data structures for composition rules, PIOs and entity sets as well as separate interface methods for T1 and T2.

Our new components have been attached to the MOSES framework by modifying the existing PDPs, which have received a T1/T2-dispatcher interface and specialized T2-policy-evaluation methods. Since Tier-2 access only relates to socket-based network communication, only the networking PEP (PEP\(_{NW}\)) had to be modified to pass the required arguments to the PDP.

C. Implementing Metapolicies

After outlining the basic architecture, we will now how the components of a metapolicy are implemented in our prototype.

1) Constructors: Each constructor \( C_i \) contains the composition rule of node \( i \) for building the metapolicy. As it is represented by a boolean expression using the algebraic operators (cf. Sec. III-A), it is implemented as a policy composition object (PCO) that is essentially derived from a general class for representing first-order logical expressions. For metapolicy configuration, these expressions are specified in an XML-based syntax similar to XACML [29]. Through a public method \( \text{eval} \), each PCO provides read access to the boolean value of its private composition term whose elements are references to PIOs or other PCOs, both stored in the T2SP Store.

Whenever a Tier-2 access is requested at the PEP, the PDP now evaluates the stored PCOs based on the group policy composition rule which is statically set to \( P_G = \Lambda P_i C_i, i \in G \) for our testing environment. The resulting value, which is the decision \( f_{P_G} \) of the metapolicy \( P_G \), is computed on runtime through the PIOs associated with the composition rules.\(^2\)

2) Policies: An individual access control policy of each node is stored locally in that node’s T2SP. For any other group member, as argued above, PIOs are stored rather than policy objects (which, in our implementation, implement the same abstract Java Interface as PIOs). While policy objects feature a light-weight matrix data structure as a representation of the local access function \( f \), PIOs store a session key for secure communication with another group member. This key is used by the Policy Negotiation Communicator (PNC) to establish an SSL socket over which policy decision messages are transferred.

Both policy objects and PIOs provide a public method called \( \text{acf} \) that takes a list of entities and an action identifier (cf. Sec. III-A) and returns a boolean value, representing a policy decision on these arguments. It is important to spell out that for any node \( i \), there is exactly one real policy object (featuring an actual access matrix) with a locally evaluated \( \text{acf} \)-method and exactly one PIO for each peer node. Consequently, all access functions \( f_j, j \neq i \) of peers are represented by communication stubs inside the respective PIOs, returning their value through the same \( \text{acf} \)-call.

Aside from the \( \text{acf} \) method, PIOs should in practice also implement a caching mechanism for earlier policy decisions to prevent excessive network communication, as any non-local policy is always evaluated on-demand. For the goals

\(^2\) A more flexible composition rule could be easily implemented by adding another interface to the PDP.
of our prototypical implementation, this feature was deemed non-critical and thus postponed.

3) Domains: Local security domains for each group member have been implemented in a straight-forward way: they are mere instances of a standard Set class that contains String-based, unique identifiers for entities. These identifiers are completely exchanged between all group members and stored in a namespace database of the T2SP Store. Next to the mere identifiers, this database also matches each entity with a group node ID where it is locally stored to support namespace resolution on distributed access requests. On any access requests, the PDP dispatcher method has to check if any requested resource is within the combined security domain \( D_G \), to determine if the T2 policy decision has to be computed in the first place. In this case, the eval-interface of the PCOs is used as described above.

4) Group Initialization: In order to prepare metapolicy-controlled spontaneous interaction, a logical group of nodes has to be formed. This is initiated by a user who can interactively select peer group members, currently implemented using a very simple group formation app that allows to enter a number of host names. These peers can be local to a connected WLAN or reachable via the Internet. Note that, since we are intercepting all network-communication via sockets, the actual communication protocol does not depend on the implementation of the framework: TCP sockets are subject to our modified PEP as well as Bluetooth sockets etc. Consequently, for a Bluetooth-based cooperation scenario, only the group formation app would have to be modified (e.g. to leverage Bluetooth-based device detection and pairing features, that may further ease user interaction).

As a last remark on metapolicy implementation, it should be clarified that correct synthesis and enforcement of the distributed access control policy is subject to mutual trust in tamper-proof reference monitors among all group members. In our environment, that could be achieved by exchanging integrity certificates on the implemented security architecture or, in highly sensitive applications, restricting group membership to mobile hardware that provides a TPM-style certificate.

D. Example

In the following, we will revisit the example from Section II to demonstrate policy synthesis among group members using our prototype system.

The actual process of policy synthesis is composed of two phases: first, all nodes broadcast the composition information to the whole group (grouping phase) in order to distribute the required knowledge for metapolicy access decisions. Afterwards, during the interactive phase, distributed resources from each node’s domain can be shared and accessed, hereby evaluating the group’s metapolicy.

1) Grouping Phase: After initialization of a group interaction through the group formation app, every node processes basically three steps: (1) Generate a PCO based on the local composition rule, generate a policy object based on the local access control policy and generate a domain object based on the entities that should be shared. These data structures are registered in the T2SP Store. (2) Create a PIO for the local policy; then send the PIO, the local PCO, and the domain information to each peer node using the PNC. (3) Enter each PIO-, PCO- and domain-message received from a peer node into the T2SP Store.

After these steps, the grouping phase is completed and any access decision involving the shared metapolicy can be evaluated.

2) Interactive Phase: If Alice from the example in Section II tries to communicate with a projector server the local domain of Bob, her presentation app sends an RPC-style message such as showSlide(s) to Bob’s IP at the server’s port number. (1) Now Alice’s PEP\_NW calls the PDP’s dispatcher interface, that detects a T2-access by querying the namespace database. (2) Within the PDP, each node’s PCO is retrieved from the T2SP Store by a call getPCO(node). By calling getPCO(node).eval(PP\_APP, PROJ\_APP, ACTION\_PUSH) for each node (arguments are resolved from the T2SP Store namespace database), the PDP fetches the boolean value of each composition term. (3) Each PCO internally evaluates the respective composition term, hereby calling several PIO objects, among them one local policy object (Alice’s own policy). All other PIOs internally call a method negotiateDecision of the PNC with the ACF’s arguments (see above), returning a boolean decision value after communication with the group node associated with the calling PIO. (4) After evaluating all ACFs, the eval method of each PCO returns a boolean for the value of its respective group member’s composition term. Using these values for all group members, the PDP finally composes \( f_{T2} \). (5) In case \( f_{T2} \) is true, the original message (showSlide(s)) is relayed to the handler interface in the Android kernel by the PEP\_NW; otherwise, a MOSES-based notification service tells the user that the app tried a forbidden group-based resource access.

As already mentioned, Bob has to trust in the tamperproof implementation of Alice’s T2 framework here – since effective policy-enforcement is exclusively carried out at the client node of each request. Since we are dealing with a decentralized system here, it is inevitable that some group member has to take responsibility for enforcing an access decision; however, there is also an advantage in client-side policy enforcement: redundant traffic by messages that already carry payload of a forbidden access (such as the showSlide(s) RPC) can be avoided this way.

E. Performance Impact

To get an understanding of the practical costs of our approach, we will present some basic observations concerning performance overhead that comes with spontaneous metapolicy enforcement, based on a non-optimized proto-
The computation time of policy evaluation rather than in message delivery time and jitter, experiments were conducted in a LAN-based environment with negligible communication overhead. Moreover, due to the qualitative goal, only virtualized instances of our modified Android system are used rather than physical devices. This allows us to compare the computation time of pure T1, MOSES-involving access decisions with that of T2 access decisions in the mentioned environment. It should be noted that the impact on energy consumption is not yet taken into account, albeit an important criteria for future studies under real-world conditions (with an implementation based on physical hardware).

We used a scenario of increasing, yet realistic group sizes from 2 to 10 member nodes, each providing one entity and the same local policy, which is to allow all group members access to the local entity. As constructors, every node \( i \) uses \( C_i = \bigwedge_P P_i, \ i \in G \). In order to keep the interpretation as clear as possible, we decided not to employ any complex rules for constructors and access control policy here — this is, however, subject of ongoing research and future studies. For each group size, a total of 100 access requests has been issued to T1 as well as T2 mechanisms, measuring the runtime from PDP invocation until the return of the decision value (“accept” or “deny”). We than compared the average values for local, pure T1 accesses and such involving the group policy (T1 plus T2). The results are shown in Table I.

As can be seen there, an overhead factor of 2 to about 7 was measured compared to pure T1 access control for up to 10 group members. Judging from the absolute runtime, the measure computation times still seem reasonable to not impair user experience in real world, given that we chose rather large group sizes for most applications. Still, it is important to keep in mind that with more complex and computational demanding composition and policy rules, the framework should exhibit a quadratic worst-case runtime based on the number of group members: the policy decision \( f_P \) requires evaluating each member’s constructor, where again each member’s policy decision may be required. In practice, it can be expected that these effects are mitigated by caching mechanisms (not implemented in our prototype) and, as already mentioned, limited group size in most application scenarios.

### V. Summary

This paper argues for an approach to a secure policy-controlled cooperation between mobile devices that interact in temporary, sporadically and spontaneously formed groups. Core of the approach are meta-policies that govern ad-hoc construction of security policies for each cooperation, consisting of an algebraic constructor, a policy description, and a policy domain specification. The precision of the algebraic constructors allows for a fully automatized construction of cooperation policies carried out efficiently and transparently within the TCB of each participating system without the need for any user interaction.

Like regular security policies that govern a mobile device, metapolicy are designed by a company’s security team and are embedded into the security architecture of a mobile device. Their simplicity allows for a fast and efficient policy construction with low runtime and memory complexity.

The resulting security architecture enforces metapolicies among a distributed, spontaneously created group of nodes. It preserves the autonomy of individual group members and tries to achieve high-performance policy communication and policy composition as well as user transparency as far as possible. Our prototypical implementation illustrates the practical feasibility of this approach. Based a qualitative comparison of performance overhead imposed by the distributed policy enforcement, we argue for promising practical employment. To support this claim with respect to a real-world scenario, the presented framework is subject to ongoing development and research.

### References
