Smart and Secure Cross-Device Apps for the Internet of Advanced Things

Christoph Busold[1], Stephan Heuser[1], Jon Rios[1], Ahmad-Reza Sadeghi[2], and N. Asokan[3]

[1] Intel CRI-SC, TU Darmstadt
[2] TU Darmstadt/CASED
[3] Aalto University and University of Helsinki
`{christoph.busold,stephan.heuser,rios.jon,ahmad.sadeghi}@trust.cased.de`,
`asokan@acm.org`

Abstract. Today, cross-device communication and intelligent resource sharing among smart devices is limited and inflexible: Typically devices cooperate using fixed interfaces provided by custom-built applications, which users need to install manually. This is tedious, time consuming, bears security and privacy risks, and contrasts the idea of Internet of Things (IoT) where intelligent devices operate in concert to enrich the overall user experience by sharing resources and capabilities.

We present Xapp, a context-aware service mobility framework for Android. Our goal is to enable users to securely distribute the functionality of applications to mutually untrusted smart devices, e.g., to enable a smartphone to use a nearby Android TV screen as a display for a video call, let a smartphone navigation app direct an autonomous vehicle, or let it use the vehicle for an object-recognition task rather than using a cloud service with the attendant privacy risks. We built a prototype for Android as the first step towards this goal. Our system is a set of extensions to the existing Remote-OSGi service platform, an emerging industry standard which unfortunately does not secure the communications between devices. This paper describes our proposal for the required security architecture. We designed and implemented an authentication protocol suite, where trust is bootstrapped using NFC for the sake of usability. On top of this we built a fine-grained access control system so that mutually mistrustful Xapp apps can be used simultaneously in the same neighborhood and even on the same devices. Hence, with Xapp users can run an Android app across multiple devices without having to install it on each of them individually. As proof of concept we present the implementation and evaluation of a video call app.

1 Introduction

Advanced embedded devices have been undergoing a dramatic development in the last decade: different classes of devices in different form factors, ranging from personal information and entertainment devices (e.g., smartphones, tablets, smart TVs, wearables, automotive head units for smart cars to industrial automation systems and sensors in smart factories, are being equipped with increasing computing, storage and wireless communication capabilities. The Internet of Things (IoT) promises to intelligently interconnect these devices where applications adapt to available resources in the environment and share their capabilities to improve the user-experience and maintainability significantly: Consider for instance placing a video call from a smartphone using a nearby Android TV [2] as a display; a smartphone navigation app using the more precise GPS sensors and larger display of the head unit available in a modern vehicle; letting a navigation app direct an autonomous vehicle, or resource-constrained devices outsourcing computationally expensive tasks (e.g., object recognition) to other more powerful devices.

However, today the ability for such intelligent and adaptive device collaboration falls short. Current network discovery and media sharing protocols, like UPnP [47], DLNA [16], Apple Airplay [5] or Samsung AllShare [40], limit themselves to a set of pre-defined services. More sophisticated use-cases for advanced device collaboration, be it in the area of smart vehicles, smart buildings or personal entertainment, require custom software components that have to be installed, managed and configured individually on each device. This is tedious, time consuming, and poses security and privacy risks. Moreover, existing solutions for collaboration among devices based on migrating code from one device to another (e.g., [37,20]) do not adequately address the security and privacy risks.

Our goal and contributions. We present Xapp, a context-aware service mobility framework for Android, which aims at enhancing resource sharing among advanced IoT devices. Our main contributions are as follows:

1. The design of a framework that enables users to securely run an Android app across multiple devices without having to install it on each of them individually.
2. The design and implementation of an authentication protocol suite where trust is bootstrapped using NFC for the sake of usability (Section 3).
3. A prototype of this framework on the service-based R-OSGi [36,39] software stack, an emerging industry standard which we extended with mechanisms for fine-grained access control and secure communication (Section 4).
4. A proof-of-concept evaluation of a video calling application built using Xapp (Section 5 and 6).

Xapp differs from prior work (Section 7) on distributed cross-device functionality in two major aspects. First, it provides fine-grained access control on sensitive resources using a lightweight token-based authentication and authorization system. Second, it allows users to keep sensitive assets on their trusted devices. By adopting standard technologies where possible, Xapp supports multiple COTS operating systems and can be deployed either as a system-centric platform component or be installed as an app without changes to the underlying operating system.

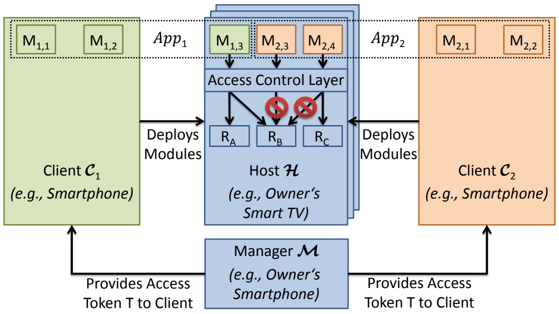

M1,1
M1,2
App1
M1,3
M2,3
M2,4
App2
M2,1
M2,2
Access Control Layer
Deploys Modules
RA
RB
RC
Deploys Modules
Client C1 (e.g., Smartphone)
Host H (e.g., Owner's Smart TV)
Client C2 (e.g., Smartphone)
Manager M (e.g., Owner's Smartphone)
Provides Access Token T to Client
Provides Access Token T to Client

Fig. 1. System Model: Entities and Interaction

2 System Model

2.1 Entities and Interactions

Our system model, presented in Figure 1, involves the following entities:

- The **Host** $\mathcal{H}$ provides resources R to other devices (e.g., a smart TV sharing its screen, camera and microphone).
- The **Manager** $\mathcal{M}$ grants access to resources R on $\mathcal{H}$ to other devices (e.g., the smartphone of the smart TV's owner).
- The **Client** $\mathcal{C}$ initiates the communication and distributes parts of its application to $\mathcal{H}$ in order to use resources R.

In our model entities are devices in a network, identified by their IP addresses resolved by using service discovery (cf. Section 7). Applications are partitioned into a set of modules M, which represent different tasks implemented by the application and depend on a set of available resources R. This module-based approach is in line with recent component-based programming models used, for instance, Android (cf. Section 4.1). We use cryptographic *access tokens* T to authenticate a client $\mathcal{C}$ to a host $\mathcal{H}$ and to define $\mathcal{C}$'s privileges to access resources R on $\mathcal{H}$. Consider a video call where a user wants to access the resources of a smart TV with her smartphone. In Figure 1 the client $\mathcal{C}_1$ (user's smartphone) requests an access token T for $\mathcal{H}$ (smart TV) as well as resources R_A and R_B (e.g., camera and microphone to place a video call) from $\mathcal{M}$ (TV owner's smartphone). The owner authorizes this request using a graphical user interface on $\mathcal{M}$. Client $\mathcal{C}_2$ similarly requests an access token T for $\mathcal{H}$ and R_C (e.g., Internet connection). They upload their respective application modules $\mathsf{M}_{1,3}$, $\mathsf{M}_{2,3}$ and $\mathsf{M}_{2,4}$ to $\mathcal{H}$. After access tokens have been issued, $\mathcal{M}$ does not have to be involved at runtime anymore. The modules on the clients $\mathcal{C}_1$ and $\mathcal{C}_2$ and the modules on $\mathcal{H}$ form the distributed applications App_1 and App_2.

2.2 Threat Model

External Adversary. The main security objective of our solution is to prevent unauthorized access from one device to sensitive resources R of another device.

External attackers $\mathcal{A}_{ext}$ are classical Dolev-Yao adversaries [17]: They do not have access to any of the devices or application modules M involved, but have full control over the network and thus can eavesdrop, manipulate, inject and replay messages. Such an attack could be used, for example, to inject malicious code into an application module, which is transmitted to another device.

Internal Adversary. Each client $\mathcal{C}$, host $\mathcal{H}$ or application module can potentially be an internal attacker $\mathcal{A}_{int}$, resulting in two possible scenarios. First, a malicious $\mathcal{C}$ can send a malicious module to $\mathcal{H}$ in order to gain unauthorized access to resources R and sensitive information, or even infect the platform or other modules M on $\mathcal{H}$. Xapp should mitigate attacks from the malicious module on $\mathcal{H}$ or any other application modules M running on it.

Second, a malicious host $\mathcal{H}$, hosting application modules M, may attempt to compromise the client application, for example by tampering with modules M running on $\mathcal{H}$. Xapp should support the developer in protecting his application against such attacks by storing and processing sensitive data only on the user's trusted device (e.g., his smartphone acting as client $\mathcal{C}$).

2.3 Objectives and Assumptions

Assumptions. Every host $\mathcal{H}$ trusts its manager $\mathcal{M}$ and vice versa. This means, $\mathcal{H}$ defers to $\mathcal{M}$ as the authority who defines access control policies for local resources R, and $\mathcal{M}$ trusts $\mathcal{H}$ to enforce these access control policies correctly. Moreover, the operating system and deployed software on $\mathcal{H}$ provide sound protection against privilege escalation attacks, i.e., we assume that a module deployed by $\mathcal{A}_{int}$ cannot bypass existing access control mechanisms.

Security Objectives. Given our assumptions our main security objective is that a user's sensitive data, applications and modules M on the user's own device (client $\mathcal{C}$), are protected from the internal adversary $\mathcal{A}_{int}$ on a host $\mathcal{H}$. Furthermore, $\mathcal{A}_{int}$ can neither compromise other sensitive applications nor modules M and their data on a connected host $\mathcal{H}$. An external attacker $\mathcal{A}_{ext}$ cannot gain access to any resource R by eavesdropping on or manipulating the network channel.

Functional Objectives. The performance overhead should be low, meaning minor user interaction and the capability to automatically move modules M to a host $\mathcal{H}$. Moreover, application modules M should run independently of the underlying hardware and operating system. This requires compatibility with common operating systems. Ideally Xapp should run as a third party application.

3 Design of Xapp

In this section we present the design of our cross-device application framework Xapp. It comprises a security architecture for sandboxing modules of different applications and stakeholders (cf. Section 3.1) and a generic resource control concept (cf. Section 3.2). Furthermore, Section 3.3 describes the token-based authentication and authorization system.

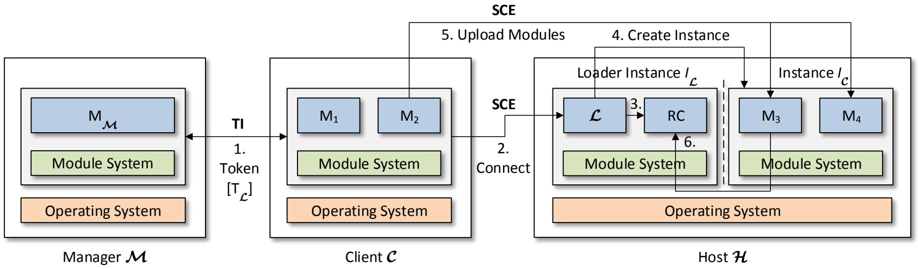


Fig. 2. Xapp Architecture

3.1 Architecture Overview and High-Level Idea

Our generic architecture is shown in Figure 2. On every host $\mathcal{H}$ a component called Loader $\mathcal{L}$ manages the modules M running on $\mathcal{H}$ and their privileges to access resources R on $\mathcal{H}$. $\mathcal{L}$ is initially installed and configured on each host, either by the owner or by the device vendor. The owner takes ownership of $\mathcal{L}$ by establishing a shared symmetric key $K_{\mathcal{M}}$ between the manager $\mathcal{M}$ (e.g., his smartphone) and $\mathcal{L}$. For our implementation we use a key agreement protocol over NFC due to the required physical proximity [23]. This approach is similar to the *resurrecting duckling model* [45], where physical contact creates a binding between two entities.

Xapp enables the developer to encapsulate the functionality of an application on a client $\mathcal{C}$ into a set of modules M, which potentially use resources on a remote host $\mathcal{H}$. We implemented an adaptation of the *extended duckling model* [44] to control which clients may upload modules to $\mathcal{H}$, and which resources may be used by a client $\mathcal{C}$. When a client $\mathcal{C}$ wants to use resources R of $\mathcal{H}$, it first requests an access token $[T_{\mathcal{L}}]$ from $\mathcal{H}$'s manager $\mathcal{M}$ (Step 1) using the Token Issuing protocol (TI). $\mathcal{C}$ authenticates to $\mathcal{L}$ using the Secure Channel Establishment protocol (SCE) with this access token (Step 2), which is forwarded to the Resource Controller (RC) (Step 3). The relevant protocols will be explained later in Section 3.3. Next, $\mathcal{L}$ creates a restricted execution environment $I_{\mathcal{C}}$ (Step 4) for modules M uploaded by $\mathcal{C}$ (Step 5). Modules which trust each other (e.g., modules belonging to the same application) may share an instance. Modules run inside their instance $I_{\mathcal{C}}$, which provides life-cycle management. $\mathcal{L}$ is executed inside a privileged instance $I_{\mathcal{L}}$ with access to all resources.

In Xapp instances are created on demand and removed when they are no longer needed, e.g., because their modules are removed. To protect $\mathcal{H}$ from malicious modules of the internal adversary $\mathcal{A}_{int}$, instances $I_{\mathcal{C}}$ follow the principle of least privilege, meaning that direct access to resources is limited to what is basically required by their modules. When a module aims to access shared resources on $\mathcal{H}$, it queries RC located inside $I_{\mathcal{L}}$ (Step 6). RC mediates access to resources R based on a Policy $P_{\mathcal{C},\mathcal{H}}$ included in the token $[T_{\mathcal{L}}]$, as described in the following section.

3.2 Resource Control Concept

When the manager $\mathcal{M}$ creates an authentication token $[T_{\mathcal{L}}]$ for the client $\mathcal{C}$, it can bind a Policy $P_{\mathcal{C},\mathcal{H}}$ to this token. Policies are forwarded by the Loader $\mathcal{L}$ to the Resource Controller RC, which is responsible for their enforcement on $\mathcal{H}$. A Policy consists of a set of individual privileges. Each privilege $\mathsf{Privilege}(\mathsf{R}, \mathcal{C}, \mathcal{H}, \mathsf{S}) = \mathit{Yes} \,|\, \mathit{No} \,|\, \mathit{Ask}$ describes whether the instance $I_{\mathcal{C}}$ may access resource R on $\mathcal{H}$, optionally limited to a given state S (e.g., time of day). The *Ask* value specifies that $\mathcal{H}$ should consult $\mathcal{M}$ at runtime when $I_{\mathcal{C}}$ tries to access this resource. Policies can further contain optional lifecycle constraints to address possible resource starvation attacks by malicious modules. For example, $\mathcal{M}$ can define that a shared resource is only accessible for a specified amount of time, or that $I_{\mathcal{C}}$ should be removed after a certain time span.

Consider the video call use case, where $\mathcal{M}$ creates a policy restricting the access of $\mathcal{C}$'s instance $I_{\mathcal{C}}$ to the camera, microphone and screen of the smart TV $\mathcal{H}$, thereby protecting the privacy of the smart TV's owner. Modules installed by $\mathcal{C}$ are denied access to other sensitive resources, such as photos accessible by the TV. Finally, $\mathcal{M}$ uses a state-aware policy to allow $I_{\mathcal{C}}$ to access the camera and microphone only when the video call module is running in the foreground on $\mathcal{H}$, and to automatically remove $I_{\mathcal{C}}$ after one hour.

3.3 Authentication and Authorization Protocols

Our design includes a flexible and secure protocol suite providing for authentication of clients, authorization for resource access and security on the communication links. This protocol suite is based on standard cryptographic primitives and due to space constraints we moved its details in Appendix A. Our protocol suite also provides offline verification, i.e., the access control token is verifiable by $\mathcal{H}$ if its manager $\mathcal{M}$ is not available. Offline verification can be achieved by token-based protocols such as Kerberos [34]. However, Kerberos requires a database with known clients, which is managed outside the protocol. Therefore we designed a custom token protocol, which can handle both ad-hoc as well as long-term clients and at the same time reduces the complexity of Kerberos.

Overview. Our protocol consists of two parts (cf. Figure 2). During the *Token Issuing Protocol* (TI) $\mathcal{M}$ issues a Token $[T_{\mathcal{L}}]$ to $\mathcal{C}$. $[T_{\mathcal{L}}]$ is bound to a key $K_{\mathcal{C}}$, which is computed through a Diffie-Hellman key agreement scheme DH between $\mathcal{M}$ and $\mathcal{C}$. $\mathcal{C}$ uses $[T_{\mathcal{L}}]$ to authenticate itself to $\mathcal{L}$ and to establish a secure channel using the *Secure Channel Establishment Protocol* (SCE). It proceeds to request a new execution instance $I_{\mathcal{C}}$. Finally, $\mathcal{C}$ uses the SCE protocol to connect to $I_{\mathcal{C}}$ by creating a new token $[T_{I_{\mathcal{C}}}]$ encrypted by K_C and with a randomly chosen key K_I inside. The only setup requirement is a shared symmetric secret key between $\mathcal{M}$ and $\mathcal{H}$, denoted $K_{\mathcal{M}}$, which is used to authenticate and encrypt tokens with the help of an authenticated encryption scheme AE. As noted in Section 3.1 $K_{\mathcal{M}}$ has been established during the initial pairing between $\mathcal{M}$ and $\mathcal{H}$.

Interactive Privilege Evaluation. As described in Section 3.3, resources protected by an *Ask* privilege require runtime consultation of the manager $\mathcal{M}$. For

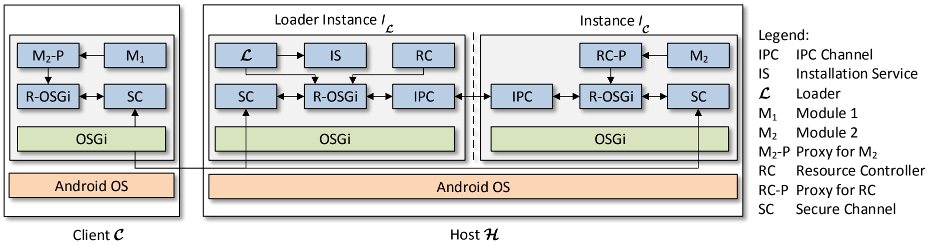


Fig. 3. Architecture of the Implementation

that purpose, the relevant host $\mathcal{H}$ sends the identity of $\mathcal{C}$ and the identifier of the requested resource R together with a nonce $\mathcal{N}$ to $\mathcal{M}$. To secure the authenticity of $\mathcal{M}$'s responses, $\mathcal{M}$ computes a message authentication code (MAC) over the decision value and the original request including the nonce $\mathcal{N}$ using the shared secret key $K_{\mathcal{M}}$. If $\mathcal{H}$ fails to verify this MAC or does not receive a response at all within a certain time frame, it defaults to deny the request.

Revocation. Since our solution focuses on time-limited deployment of cross-device applications via lifecycle constrains (cf. Section 3.2) we do not consider revocation in our current implementation. However, token revocation could be added to Xapp by means of revocation lists. The integrity and authenticity of revocation lists can be assured using MACs based on a key derived from $K_{\mathcal{M}}$. Alternatively, we could adopt a token status protocol comparable to OCSP [41], but since Xapp is designed for offline token validation we deem revocation lists to be better suited.

4 Implementation

Our implementation is based on the Apache Felix OSGi [3,36] framework and the R-OSGi RPC layer [39]. We run our framework on Android, which serves as an example of a modern operating system for advanced IoT devices. We highlight the technical challenges we had to tackle and describe several security extensions we developed for R-OSGi. Figure 3 shows the instantiated components.

4.1 Platform Considerations

Module System. We implemented the module system is on the OSGi platform [36], a widely-deployed platform-independent industry standard for software modularization. OSGi allows us to easily integrate existing solutions that can extend our framework with further desired functionality (such as service discovery protocols [29,47,4]), which is orthogonal to our work. OSGi divides applications into modules, called *bundles*. A bundle is a collection of self-contained Java packages, arbitrary data and a manifest file. This manifest contains meta data about the bundle along with its platform requirements, provided services and dependencies on other bundles. At runtime, bundles interact via services, which can be published to and consumed by other bundles.

Remote OSGi. The adoption of OSGi enables us to seamlessly connect modules on different devices using the remote service layer of Remote OSGi (R-OSGi) [39]. R-OSGi extends the concept of services in OSGi to remote services, which can be published to and accessed from other framework instances, possibly running on different machines. At runtime, R-OSGi can connect to other hosts running R-OSGi and search them for available remote services.

Target Operating System. The platform independence of OSGi allows us to instantiate our framework on a wide range of operating systems for advanced IoT devices with different capabilities, ranging from mobile devices and automotive head units to desktop PCs and virtual machines in cloud environments. Individual security aspects of the target operating system (most importantly application isolation and access control) must be considered when adopting our framework. For example, Android relies on process-level permissions and per-app sandboxes.[4]

For our prototype implementation we selected Android as the target platform not only because it is the most popular platform for smartphones and tablets [19], but also because it is deployed in other IoT market segments, e.g., automotive [1] and home entertainment [2]. While documentation on Android Auto is currently limited, Android TV is a standard Android distribution optimized for large screens and thus allows the deployment of Xapp without further modifications. Android is based on a Linux kernel and executes every app inside a Java virtual machine running within a separate process under its own Linux User Id (UID), which is set at installation time. Hence different applications are sandboxed at operating system level. Furthermore, Android enforces a permission framework on processes, which restricts access to system services and resources like network, file system or sensors.

4.2 Loader

The Loader $\mathcal{L}$ (see Figure 3) is $\mathcal{H}$'s interface to an external client $\mathcal{C}$ and exposes its functionality to remote and local application modules via R-OSGi remote services. This allows clients to create, remove, start and stop their instances on a host and to deploy application modules, as explained in the following. The Loader is implemented as a set of OSGi bundles running inside a privileged instance $I_{\mathcal{L}}$.

Installer Service. The Installer Service (IS, cf. Figure 3) is used by $\mathcal{L}$ to create and remove client instances on $\mathcal{H}$. While the implementation of IS is platform-specific, it communicates using a standardized OSGi service interface with the

---

[4] On PCs, IBM's Java JVM 8 provides a multi-tenancy environment [26], which efficiently isolates Java applications executed in one Java VM and uses the Java Security Manager [27] for access control. Another approach particularly interesting in the context of cloud-based environments is the GuestVM project [46], which provides isolated Java runtime environments on top of the Xen hypervisor.

platform-agnostic Loader component. On Android, instances are implemented as Android applications and isolated by Android's sandboxing mechanism (cf. Section 4.1). Our Installer Service IS for Android uses a base template application in the form of an Android Application Package (APK), which includes the OSGi framework and required bundles to communicate with $\mathcal{C}$ (e.g., R-OSGi). Android apps are identified by a unique package name. Accordingly, IS rewrites the package file with a new name and configures parameters specific to the new instance $I_{\mathcal{C}}$, such as the listening port of R-OSGi.

The Loader $\mathcal{L}$ can be distributed by the device manufacturer as an Android system app or installed by the user as a standard Android app on a host $\mathcal{H}$. When app modules are deployed by a client $\mathcal{C}$, the installation of the client-specific instance $I_{\mathcal{C}}$ is ideally performed silently without user interaction once $\mathcal{H}$ has validated $\mathcal{C}$'s access token. Due to Android-specific limitations this is only possible when $\mathcal{L}$ is an Android system app: For security reasons third-party apps cannot install or uninstall other applications silently on stock Android. Thus, if the Loader is installed as a standard third-party app on a stock Android device, our framework requires minimal user interaction, since the user has to approve the installation of $I_{\mathcal{C}}$ by clicking a button on $\mathcal{H}$.

Resource Controller. The Resource Controller (RC) exposes resources of the host $\mathcal{H}$ to a client's instance $I_{\mathcal{C}}$ using a R-OSGi service. It is executed inside the privileged Loader instance $I_{\mathcal{L}}$, which holds all permissions required to access the resources R (e.g., contacts or camera) exposed to instances I. Access to resources is mediated according to the instance-specific access control policy defined by $\mathcal{M}$ and contained in the token $[T_{\mathcal{L}}]$. The implementation of the Resource Controller is platform-specific, while its interface is the same on different operating systems.

In general, there are two possible approaches to implement access control on resources: Either the Resource Controller RC uses the existing platform-specific access control mechanisms, or RC implements the required access control hooks itself. Both approaches have advantages and disadvantages:

In the former case, RC maps privileges to operating system specific access control mechanisms. For example, Android uses *permissions* for access control on APIs as well as *discretionary* and *mandatory* access control [43] for kernel-level resources. More advanced architectures [9,24] provide interfaces to programmatically influence system-level access control decisions at runtime and could potentially be integrated with Xapp. However, such an integration would limit our solution and violate our interoperability requirements.

In the latter case, RC implements access control on resources itself, also considering fine-grained and state-aware access control policies. We opted for this approach in our implementation, since it does not require changes to the underlying operating system.

4.3 Our Extensions to Remote OSGi

Our implementation provides several security extensions to the Remote OSGi (R-OSGi) framework, as described in the following.

Secure Network Channel. The R-OSGi middleware (cf. Section 4.1) offers by default only TCP communication and provides no protection against an external attacker $\mathcal{A}_{ext}$. To enable secure communication between the Loader instance $I_{\mathcal{L}}$, the client $\mathcal{C}$ and his remote instance $I_{\mathcal{C}}$, we extended R-OSGi with a secure network channel (SC), which provides both confidentiality and integrity using authenticated encryption with a symmetric key (see SCE protocol in Appendix A).

Local IPC Channel. To provide efficient communication between different OSGi frameworks in separate sandboxed instances on the same host $\mathcal{H}$ (e.g., $I_{\mathcal{C}}$ and $I_{\mathcal{L}}$), we implemented an IPC communication channel using domain sockets.

Channel Filter. Since the original design of R-OSGi does not differentiate between different network channels, a service can only decide whether it wants to be published to remote hosts or not, and in that case it is always registered on *all* available channels. This is insufficient, if one wants to expose a service only via IPC to local instances executed on the same host $\mathcal{H}$. Therefore, we modified the R-OSGi implementation to include a filter on communication channels, so that services can choose the channels they are available on.

Endpoint Identification. Another challenge is that services do not know over which channel and from which endpoint they were called (i.e., remotely or locally), because this connection between the function call and the originating channel endpoint is hidden by the abstraction of R-OSGi. In our model, this information is crucial in order to decide whether access to a service should be granted or not, depending on the identity of the caller (i.e., modules M of a client $\mathcal{C}$ executed in instance $I_{\mathcal{C}}$). Thus we implemented a function to retrieve the identity of the current caller from R-OSGi. For the IPC channel we get the Linux UID of the connected process and in case of the secure network channel we extract the identity from the token that was used during the channel establishment protocol.

4.4 Xapp Development Model

To support Xapp, apps need to adhere to the (R-)OSGi programming model, since Xapp is not limited to one specific operating system. Specifically, developers need to integrate an OSGi runtime environment into their applications, such as the open-source implementation *Apache Felix* [3], on top of which the Xapp bundles (mainly R-OSGi and the Loader) as well as application-specific bundles are executed. Consequently, application modules which should migrate between hosts need to be implemented as OSGi bundles. These bundles at runtime communicate with other bundles on the local or remote host via OSGi services. OSGi services are comparable to Android services in that they adhere to the same RPC communication paradigm. To ease the work of developers who want to adopt our solution, Xapp provides a set of service definitions for common resources, such as *contacts information*, *camera* and *microphone.*

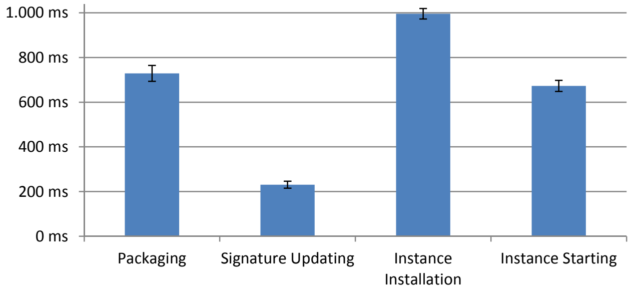


Fig. 4. Performance Evaluation (Basic Instance)

5 Evaluation

To evaluate our implementation we used two Samsung Galaxy S3 I9300 smartphones running Android 4.0.4 (client and manager) and a Nexus 10 Tablet running Android 4.2.1 (host) connected over a 802.11bgn wireless network. We use the industry standard algorithms AES-256 in EAX mode as authenticated encryption scheme AE and ECDH-256 as key exchange protocol DH.

Performance. We evaluated the performance of different components in our solution. For the Android-based implementation of the token issuing protocol we measured the elapsed time between starting the communication with the manager and receiving the token $[T_{\mathcal{L}}]$. Overall the protocol takes 308.28 ± 27.73 milliseconds on average over 20 runs.

For our framework we first performed microbenchmarks to measure the time required for creating a basic instance containing no bundles. This includes all steps starting from verifying the access token $[T_{\mathcal{L}}]$, creating the application package, installing and finally starting the instance I. The results are presented in Figure 4 and show the average and the standard deviation of the time required to perform all steps over 20 runs. These numbers are reasonable considering that our implementation has not been optimized for performance yet, and we refer to our case study below for further discussion of these results. Further, we verified that the OSGi framework incurs no noticeable performance overhead at runtime using the Java Linpack benchmark [32] both in a regular Android app and in a Xapp module. The average performance over 20 runs is 143.15 ± 0.13 MFlops and 137.41 ± 0.33 MFlops respectively, which shows a small difference of 4.18%.

We also performed microbenchmarks to measure the performance impact introduced by our access control architecture. In our Android-based implementation we query the contacts database to retrieve a single contact both in a regular Android app and in a Xapp module. The process takes on average 17.47 ± 4.41 and 65.50 ± 3.86 milliseconds respectively over 1000 runs. The high standard deviation is caused by varying system load. The difference of around 48 milliseconds introduced by the redirection of calls via the Resource Controller RC and the access control enforcement is partially caused by marshalling the data over the domain socket. The overhead can be reduced by mapping the memory into the process, for example using Binder IPC, instead of copying it.

Interoperability and Portability. Our design enables the isolation of modules deployed on any operating system and hardware platform which provide adequate sandboxing and privilege separation capabilities. Since our framework operates on the application level, it requires no changes to the operating system, as demonstrated by our implementation on Android. When an Android device vendor deploys Xapp, it is even possible to install new instances without user interaction by installing the Loader as a *system* app (see Section 4.1).

It should be noted that while we instantiated our framework on Android, our architecture only requires a standard-compliant Java Runtime Environment with an OSGi framework and a platform-dependent isolation and privilege separation mechanism (cf. Section 4.1). Java is used on a variety of operating systems and platforms, from smart mobile devices to mainframes, and open-source implementations of the Java Virtual Machine and a of different OSGi implementations are available.

Usability. Pairing of devices via NFC has been adopted for a wide range of consumer devices, such as printers and Bluetooth speakers. Our performance measurements (cf. Figure 4) indicate that the time required to deploy app modules on one or more hosts (cf. Figure 4) is reasonable considering the alternative, which is to manually search, install and later uninstall an app on each host. While the definition of access control rules in the manager app is straightforward, one possible limitation is that with a growing number of rules a user might be tempted to always allow any requests for access to privileges by a client [18]. However, since the functionality of app modules is limited and tailored to specific use cases, they only need access to a very limited set of resources, which limits the number of privileges a manager has to consider.

Proof of Concept: Video Call Application. To demonstrate the advantages and feasibility of our solution we implemented the video calling use case, where Alice uses her smartphone (Client $\mathcal{C}$) and Hector's Smart TV (Host $\mathcal{H}$) to place a video phone call to Bob. This use case highlights an advantage of app partitioning over just connecting the TV to the phone: The video stream does not have to be routed through Alice's smartphone, but can be processed and sent to the TV directly by Bob's smartphone. Furthermore, Xapp protects Alice's privacy in case the TV is untrusted, since Alice does not have to enter her login credentials on the potentially malicious smart TV. Instead, she can keep sensitive data (e.g., login credentials or contact information) on her trusted device (her smartphone).

During the implementation and evaluation of Xapp we involved a team of eight students and staff members who performed preliminary usability tests by initiating a call between two Android smartphones using a nearby smart TV (represented by a Nexus 10 tablet, see Section 5). Currently we are working on a more extensive and representative usability study.

Table 1 presents the performance measurement results for creating an instance within this use case. In contrast to the basic instance above, these numbers contain a transmission phase, where modules with an overall size of 34.2

Step	Average Time (ms)
Signature Updating	330.86 ± 12.86
Instance Installation	1,122.16 ± 32.47
Instance Startup	2,228.47 ± 48.49
Bundle Transmission	1,837.77 ± 49.79
Repackaging	1,271.20 ± 81.16
Total	6,790.46 ± 106.43

Table 1. Performance Evaluation (Case Study)

KByte are sent to the host and added to the installation package, which increases the startup time. The total time of our unoptimized case study implementation takes about 6.79 seconds to deploy the relevant app modules on a host $\mathcal{H}$, which is comparable to downloading and installing apps via an app market. Note that a client $\mathcal{C}$ can deploy modules on multiple hosts in parallel and in contrast to classic applications our cross-device apps do not require any further lifecycle management such as updates and configuration on the involved devices.

6 Security Considerations

In this section we discuss how Xapp achieves the previously defined security goals (cf. Section 2.3).

External Adversary. An external adversary $\mathcal{A}_{ext}$ needs valid access tokens to gain access to either the loader $\mathcal{L}$ on host $\mathcal{H}$ or an instance $I_{\mathcal{C}}$ deployed by a client $\mathcal{C}$. The initial pairing between $\mathcal{H}$ and the manager $\mathcal{M}$, during which a shared symmetric key $K_{\mathcal{M}}$ is established, is performed through confidential and authenticated communication. In our implementation we establish this key over NFC which is resistant against man-in-the-middle attacks due to the required physical proximity [23]. Without knowledge of the cryptographic key $K_{\mathcal{M}}$, $\mathcal{A}_{ext}$ cannot generate a valid access token $[T_{\mathcal{L}}]$ for $\mathcal{L}$. Similarly the properties of NFC also protect the authenticity of $\mathcal{M}$ and $\mathcal{C}$ when $\mathcal{M}$ issues a confidentiality-protected token $[T_{\mathcal{L}}]$ to $\mathcal{C}$. Without access to the key $K_{\mathcal{C}}$ stored in the token $[T_{\mathcal{L}}]$ $\mathcal{A}_{ext}$ is unable to deploy modules on $\mathcal{H}$. At runtime, $\mathcal{C}$ and $I_{\mathcal{C}}$ communicate through authenticated and end-to-end encrypted channel. These properties are bootstrapped from the access token $[T_{\mathcal{L}}]$ issued to $\mathcal{C}$ by $\mathcal{M}$, which prevents $\mathcal{A}_{ext}$ from communicating with the deployed instance.

Internal Adversary. As noted in Section 2.2 either a client $\mathcal{C}$ or a host $\mathcal{H}$ can act as an internal adversary $\mathcal{A}_{int}$. On the one hand, $\mathcal{H}$ has to be protected from a malicious module deployed by $\mathcal{C}$. To this end, we designed an access control model that mediates which modules M may access sensitive resources R on $\mathcal{H}$. To implement this model the host operating system needs to run modules M deployed in $\mathcal{C}$'s instance $I_{\mathcal{C}}$ in an isolated least-privilege container. Our Android-based implementation uses the default UID-based sandboxing mechanism (cf. Section 4.1), which effectively prevents modules M from accessing sensitive resources R directly. Instead, Xapp modules M use the Resource Controller RC

as a deputy who enforces the access control policy defined by the manager $\mathcal{M}$. The policy is protected by our token-based authentication and access control scheme, which ensures that it cannot be forged or modified by an internal adversary $\mathcal{A}_{int}$. Dynamic access control queries evaluated by $\mathcal{M}$ at runtime are protected against impersonation, modification and replay by message authentication codes with nonces.

On the other hand, sensitive resources R of a client $\mathcal{C}$ need to be protected from a malicious host $\mathcal{H}$. Xapp's module system encourages developers to enclose sensitive operations in separate modules. A client can decide where these modules are executed. Thus Xapp allows clients to ensure that modules accessing or storing sensitive data, such as long-term credentials or contact information, remain on their trusted devices. Of course the adversary could still exploit software errors, hidden backdoors or bad application design, but this risk is not higher than for traditional applications.

Discussion. To implement our access control model we rely on the integrity and security of (system) software on the host (see Section 2.3). This requirement is inherent to the solutions that operate purely on the application layer. Obviously the internal adversary $\mathcal{A}_{int}$ could extend its privileges at runtime if he could compromise any privileged system services. Furthermore, access control solutions at the application layer, such as Xapp, cannot provide resilience against confused deputy [22] or collusion [42] attacks. For example, malicious modules deployed by different stakeholders could combine their privileges and use inter-process communication (IPC) to exchange sensitive assets. Reliable control on IPC would require an extension of the underlying operating system [24,9], which is possible but does not conform to our interoperability requirement. Moreover, we note that attacks using side channels (e.g., [42]) are out of scope of our framework. We stress that these limitations apply to manually deployed applications as well.

7 Related Work

Service-based Frameworks. Related work has proposed service-based architectures to orchestrate components of distributed applications on mobile and embedded devices [13] [30] [6] [14], which mainly focus on aspects we consider orthogonal to our work, such as context-aware service composition and discovery. While Rellermeyer et al describe the general applicability of their R-OSGi framework [39] in IoT scenarios [38], Preuveneers et al [37] propose a service mobility framework for mobile devices, which enables dynamic service migration. Their approach considers state transfer and synchronization, service discovery and resource constraints, while Xapp focuses on security by allowing modules to migrate between platforms in a controlled fashion subject to strict access control.

Goncalves et al [20] propose a service mobility framework for Android, which focuses on QoS and realtime requirements for the migration of stateful services. They extend standard Android application components to be applicable in distributed systems and enable component migration between devices. While their approach is limited to the Android OS and requires developers to split apps a-priori into separate packages, Xapp assembles platform-specific instances automatically at runtime and supports fine-grained access control on resources.

Computation Offloading. Cuervo et al [11] and Kosta et al [31] use application partitioning to offload computational tasks from mobile devices to the cloud to speed up computations and reduce battery consumption. In both system models, the cloud is inherently trusted, and security aspects are out of scope. Haerick et al [21] propose an OSGi-based platform designed to outsource energy-intensive computations from mobile devices to other platforms.

Xapp is a more general approach, designed to allow different devices to collaborate and use resources and services in a distributed environment, where computation offloading is only one possible use case. Xapp provides better usability by adopting ad-hoc NFC-based trust establishments. Furthermore, our solution is designed to consider the security requirements of different stakeholders by isolating application components and by controlling access to shared resources.

Fine-Grained Access Control. Our access control scheme is similar to the approach presented by Jeon et al [28]. Their solution redirects calls to APIs via a proxy app which enforces its own fine-grained access control model, while Xapp binds capabilities to cryptographic tokens in a distributed environment.

Several system-centric security extensions have been proposed for Android, ranging from enhancements to the coarse-grained permission system [33,10], mocking privacy-sensitive data at runtime [48,7,25] to integrating mandatory access control [9,8,35,24]. A combination of extensible system centric security solutions (e.g., Flaskdroid [9] or ASM [24]) with Xapp would allow developers to adhere to Android's standard permission system and APIs. However, these solutions require modifications to the operating system.

8 Conclusion

Computing in personal, commercial and industrial environments is undergoing a paradigm shift. The advent of the Internet of Things (IoT) enables new use cases, in which classical computing platforms, smartphones and tablets, wearables and further electronic devices operate in concert. Application lifecycle management and secure resource sharing become increasingly important aspects in this area.

To address these new challenges, we present the design and implementation of Xapp, a framework for smart and secure cross-device IoT applications for Android. With Xapp, Android apps can run distributed on different devices without the need to install them manually on each device. We present a proof-of-concept implementation for a video call use case, and are currently extending this work in several directions, such as prototyping other use cases, and incorporating automatic code-partitioning techniques to provide flexible tools to developers.

Acknowledgements

This work has been co-funded by the European Project FutureID. We would like to thank Ross Anderson for his feedback that guided the paper's final revisions.

References

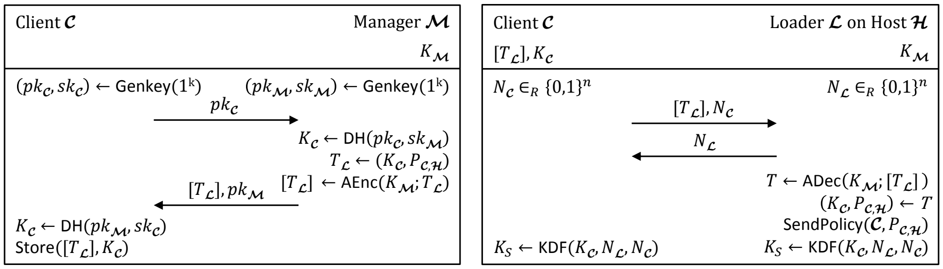
1. Android Auto. `http://www.android.com/auto/`.
2. Android TV. `http://www.android.com/tv/`.
3. Apache Felix. `http://felix.apache.org/`.
4. Apache Felix UPnP. `http://felix.apache.org/site/apache-felix-upnp.html`.
5. Apple Airplay. `http://www.apple.com/de/airplay/`.
6. S. Arbanowski, P. Ballon, K. David, O. Droegehorn, H. Eertink, W. Kellerer, H. van Kranenburg, K. Raatikainen, and R. Popescu-Zeletin. I-centric communications: personalization, ambient awareness, and adaptability for future mobile services. *Communications Magazine, IEEE*, 42(9):63–69, Sept 2004.
7. A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *HotMobile*, 2011.
8. S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and Lightweight Domain Isolation on Android. In *SPSM*, 2011.
9. S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security*, 2013.
10. M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. In *ISC*, 2010.
11. E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *MobiSys*, 2010.
12. Q. Dang. *Recommendation for Existing Application-Specific Key Derivation Functions*. NIST, 2010.
13. S. de Deugd, R. Carroll, K. Kelly, B. Millett, and J. Ricker. Soda: Service oriented device architecture. *Pervasive Computing, IEEE*, 5(3):94–96, July 2006.
14. L. de Souza, P. Spiess, D. Guinard, M. Khler, S. Karnouskos, and D. Savio. Socrades: A web service based shop floor integration infrastructure. In C. Floerkemeier, M. Langheinrich, E. Fleisch, F. Mattern, and S. Sarma, editors, *The Internet of Things*, volume 4952 of *Lecture Notes in Computer Science*, pages 50–67. Springer Berlin Heidelberg, 2008.
15. W. Diffie and M. Hellman. New Directions in Cryptography. *Information Theory, IEEE*, 1976.
16. Digital Living Network Alliance. `http://www.dlna.org/`.
17. D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *Information Theory, IEEE*, 1983.
18. A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.
19. Gartner Says Worldwide Tablet Sales Grew 68% in 2013. `http://www.gartner.com/newsroom/id/2674215`.
20. J. Goncalves, L. L. Ferreira, L. M. Pinho, and G. Silva. Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems. In *EUC*, 2010.
21. W. Haerick, T. Wauters, C. Develder, F. D. Turck, and B. Dhoedt. Transparent resource sharing framework for internet services on handheld devices. *Annals of Telecommunications*, 2010.
22. N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

23. E. Haselsteiner and K. Breitfuss. Security in Near Field Communication. *RFIDSec*, 2006.
24. S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *USENIX Security Symposium*, 2014.
25. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *ACM CCS*, 2011.
26. Introduction to Java multitenancy. `http://www.ibm.com/developerworks/java/library/j-multitenant-java/index.html`.
27. SecurityManager (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html`.
28. J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *SPSM*, 2012.
29. jSLP - Java SLP (Service Location Protocol) Implementation. `http://jslp.sourceforge.net/`.
30. J. King, R. Bose, H.-I. Yang, S. Pickles, and A. Helal. Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 630–638, Nov 2006.
31. S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading. In *INFOCOM*, 2012.
32. Linpack Benchmark – Java Version. `http://www.netlib.org/benchmark/linpackjava/`.
33. M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *AsiaCCS*, 2010.
34. B. C. Neuman and T. Tso. Kerberos: an authentication service for computer networks. *Communications Magazine, IEEE*, 1994.
35. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC*, 2009.
36. OSGi Alliance. OSGi Service Platform Release 4. `http://www.osgi.org/Main/HomePage`.
37. D. Preuveneers and Y. Berbers. Context-driven migration and diffusion of pervasive services on the OSGi framework. *IJAACS*, 2010.
38. J. Rellermeyer, M. Duller, K. Gilmer, D. Maragkos, D. Papageorgiou, and G. Alonso. The software fabric for the internet of things. In C. Floerkemeier, M. Langheinrich, E. Fleisch, F. Mattern, and S. Sarma, editors, *The Internet of Things*, volume 4952 of *Lecture Notes in Computer Science*, pages 87–104. Springer Berlin Heidelberg, 2008.
39. J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: distributed applications through software modularization. In *Middleware*, 2007.
40. Samsung Allshare. `http://developer.samsung.com/allshare-framework/technical-docs/FAQ`.
41. S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), June 2013.
42. R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society, 2011.

43. S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of NDSS*, 2013.
44. F. Stajano. The resurrecting duckling – what next? In *Revised Papers from the 8th International Workshop on Security Protocols*, pages 204–214, London, UK, UK, 2001. Springer-Verlag.
45. F. Stajano and R. J. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the 7th International Workshop on Security Protocols*, pages 172–194, London, UK, UK, 2000. Springer-Verlag.
46. The Guest VM Project. `https://kenai.com/projects/guestvm`.
47. Universal Plug-and-Play. `http://www.upnp.org/`.
48. Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *TRUST*, 2011.

A Protocols

As explained in Section 3.3, the protocols assume a shared symmetric secret key between $\mathcal{M}$ and $\mathcal{H}$, denoted $K_{\mathcal{M}} \in \{0,1\}^n$, which is used to authenticate and encrypt tokens with the help of an authenticated encryption scheme $\mathsf{AE} = (\mathsf{AEnc}, \mathsf{ADec})$, where n is a security parameter.


Client $\mathcal{C}$
Manager $\mathcal{M}$
$K_{\mathcal{M}}$
$(pk_{\mathcal{C}}, sk_{\mathcal{C}}) \leftarrow \mathsf{Genkey}(1^k)$
$(pk_{\mathcal{M}}, sk_{\mathcal{M}}) \leftarrow \mathsf{Genkey}(1^k)$
$pk_{\mathcal{C}}$
$K_{\mathcal{C}} \leftarrow \mathsf{DH}(pk_{\mathcal{C}}, sk_{\mathcal{M}})$
$T_{\mathcal{L}} \leftarrow (K_{\mathcal{C}}, P_{\mathcal{C},\mathcal{H}})$
$[T_{\mathcal{L}}] \leftarrow \mathsf{AEnc}(K_{\mathcal{M}}; T_{\mathcal{L}})$
$[T_{\mathcal{L}}], pk_{\mathcal{M}}$
$K_{\mathcal{C}} \leftarrow \mathsf{DH}(pk_{\mathcal{M}}, sk_{\mathcal{C}})$
$\mathsf{Store}([T_{\mathcal{L}}], K_{\mathcal{C}})$
Client $\mathcal{C}$
Loader $\mathcal{L}$ on Host $\mathcal{H}$
$[T_{\mathcal{L}}], K_{\mathcal{C}}$
$K_{\mathcal{M}}$
$N_{\mathcal{C}} \in_R \{0,1\}^n$
$N_{\mathcal{L}} \in_R \{0,1\}^n$
$[T_{\mathcal{L}}], N_{\mathcal{C}}$
$N_{\mathcal{L}}$
$T \leftarrow \mathsf{ADec}(K_{\mathcal{M}}; [T_{\mathcal{L}}])$
$(K_{\mathcal{C}}, P_{\mathcal{C},\mathcal{H}}) \leftarrow T$
$\mathsf{SendPolicy}(\mathcal{C}, P_{\mathcal{C},\mathcal{H}})$
$K_S \leftarrow \mathsf{KDF}(K_{\mathcal{C}}, N_{\mathcal{L}}, N_{\mathcal{C}})$
$K_S \leftarrow \mathsf{KDF}(K_{\mathcal{C}}, N_{\mathcal{L}}, N_{\mathcal{C}})$

(a) TI (b) SCE

Fig. 5. Token Issuing Protocol (TI) and Secure Channel Establishment Protocol (SCE)

Token Issuing Protocol. The Token Issuing Protocol (TI) is shown in Figure 5(a). Both $\mathcal{M}$ and $\mathcal{C}$ generate a new asymmetric key pair. $\mathcal{C}$ sends its public key $pk_{\mathcal{C}}$ to $\mathcal{M}$ over an out-of-band channel. Generally we require that this channel is integrity-protected at least in one direction ($\mathcal{C}$ to $\mathcal{M}$), so that it is immune to man-in-the-middle attacks where an attacker attempts to replace $pk_{\mathcal{C}}$ with a different public key. We use Near-Field Communication (NFC), which directly allows $\mathcal{M}$ to verify the identity of $\mathcal{C}$ due to the physical proximity required for NFC. However, alternative implementations are also feasible, for example using QR codes. $\mathcal{M}$ creates a new Token $[T_{\mathcal{L}}]$, which contains the client key $K_{\mathcal{C}}$, as well as a description of $\mathcal{C}$'s privileges on $\mathcal{H}$, denoted by the Policy $P_{\mathcal{C},\mathcal{H}}$. $K_{\mathcal{C}}$ is derived using a key agreement scheme DH (e.g., Diffie-Hellmann [15]) between $\mathcal{C}$ and $\mathcal{M}$. Finally, $\mathcal{M}$ sends the token to $\mathcal{C}$ together with its public key $pk_{\mathcal{M}}$.

Secure Channel Establishment. The client $\mathcal{C}$ uses the Secure Channel Establishment Protocol (SCE) to connect to the Loader $\mathcal{L}$ as shown in Figure 5(b): $\mathcal{C}$ sends $[T_{\mathcal{L}}]$ and a randomly chosen nonce $N_{\mathcal{C}}$ to $\mathcal{L}$. $\mathcal{L}$ decrypts $[T_{\mathcal{L}}]$ using $K_{\mathcal{M}}$, thereby verifying its integrity due to the authenticated encryption. Next, $\mathcal{L}$ extracts $K_{\mathcal{C}}$ and the Policy $P_{\mathcal{C},\mathcal{H}}$, which is forwarded to the Resource Controller RC. $\mathcal{L}$ stores $K_{\mathcal{C}}$ securely in a database and later provides a decryption service to instances of $\mathcal{C}$, so that $K_{\mathcal{C}}$ cannot be exfiltrated by modules in $I_{\mathcal{C}}$. Then, $\mathcal{L}$ generates a random nonce $N_{\mathcal{L}}$, which it sends back to $\mathcal{C}$. Finally, both sides compute a shared secret session key $K_{\mathcal{S}} = \mathsf{KDF}(K_{\mathcal{C}} \mid\mid N_{\mathcal{L}} \mid\mid N_{\mathcal{C}})$ using a suitable key derivation function KDF [12]. In our implementation we use an HMAC/SHA1-based key derivation function. After this step, the secure channel establishment is completed and $K_{\mathcal{S}}$ will be used to protect all further communication.

$\mathcal{C}$ also uses SCE to connect to $I_{\mathcal{C}}$. Therefore $\mathcal{C}$ creates a new token $[T_{I_{\mathcal{C}}}]$ with a randomly chosen key K_I. Since no policy is established between the client and the host it attaches an empty dummy policy and encrypts the complete token $[T_{I_{\mathcal{C}}}]$ with $K_{\mathcal{C}}$. On the client side, $I_{\mathcal{C}}$ decrypts $[T_{I_{\mathcal{C}}}]$ using the decryption service provided by $\mathcal{L}$.