Thingtegrity: A Scalable Trusted Computing Architecture for the Internet of Things

Tobias Rauter, Andrea Höller, Johannes Iber, Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
{tobias.rauter, andrea.hoeller, johannes.iber, christian.kreiner}@tugraz.at

Abstract

Remote attestation is used to prove the integrity of one system (prover) to another (challenger). The prover measures its configuration and transmits the result to the challenger for verification. Common attestation methods lead to complex configuration measurements (e.g., hash of all executables), which are updated every time one of the software modules changes. The updated configuration has to be distributed to all possible challengers since they need a reference to enable the verification. Recently, an idea of reducing the complexity of the configuration measurement by taking into account privileges of software modules has been presented. However, this approach has not been exhaustively analyzed since, as yet, no implementation exists. Especially in the Internet of Things (IoT) domain, where resources are constrained strictly while devices are potentially physically exposed to adversaries, attestation methodologies with reduced overhead are desireable. In this work we combine binary-, property- and privilege-based remote attestation to integrate a trusted computing architecture transparently into IoTivity, an existing IoT middleware. As a first step, we aim to enable to attestation of the integrity of complex devices with different services to constrained devices. With the help of an illustrative simulated environment, we show that our architecture reduces the effort of bootstrapping trusted relations, as well as updating single modules in the whole system, even if software and devices from different vendors are combined.

1 Introduction

Studies predict the prevalence of connected devices in the near future and estimate that there will be over 13 billion devices by 2020 [1]. Essentially, these devices are connected sensors or actuators that measure or modify their environment. The high density of sensors potentially implies privacy issues whilst the ability to access actuators from anywhere may allow adversaries to control critical infrastructure. Recently, large scale TV manufacturers are warning their customers not to discuss private information in front of their devices [2], light bulbs reveal the owner's WiFi credentials [3] and pacemakers have been controlled by unauthenticated devices [4]. Individuals are not the only target of adversaries. Supervisory Control and Data Acquisition (SCADA) systems are also continuously attacked [5]. Therefore, a lot of research has been done to improve the authentication of devices and the integrity and confidentiality of their communication. However, even if a communication partner is authenticated, how is it possible to ensure that the software running on it is not harmful?

Remote attestation is used to assure the integrity of one system (prover) to another (challenger). In order to achieve this, the prover measures its configuration and cryptographically proves the integrity of this measurement with hardware components like a Trusted Platform Module (TPM) [6] or secure co-processors [7]. The integrity of the prover's configuration is verified by comparing the measurement against a known value. The challenger therefore has to know all possible 'good' configurations of the prover. In the resource constrained embedded domain this technology has become a wide research topic for different applications and at different levels of abstraction [8, 9, 10, 11].

Today's systems are often comparable to the architecture illustrated in Figure 1a. On the one hand, different types of small devices are used for a particular purpose (e.g., a sensor or an actuator). These devices are often constrained with respect to energy and performance. On the other hand, central stations such as gateways, field controllers or powerful consumer electronics exist. These devices benefit from hardware that is becoming increasingly powerful. Consequently, it is desirable to integrate different services into one device in order to reduce hardware cost. Such devices may control actuators based on sensor values or just connect different segments in a bigger network to reduce the clusters to maintainable sizes (i.e., gateways).

Whenever communication occurs, the corresponding device needs to ensure the integrity of the central station. Here, the integration of different services on the central device causes problems. Each sensor/actuator has to know a reference configuration that is composed of all services running

International Conference on Embedded Wireless Systems and Networks (EWSN) 2016
15–17 February, Graz, Austria

ISBN: 978-0-9949886-0-7

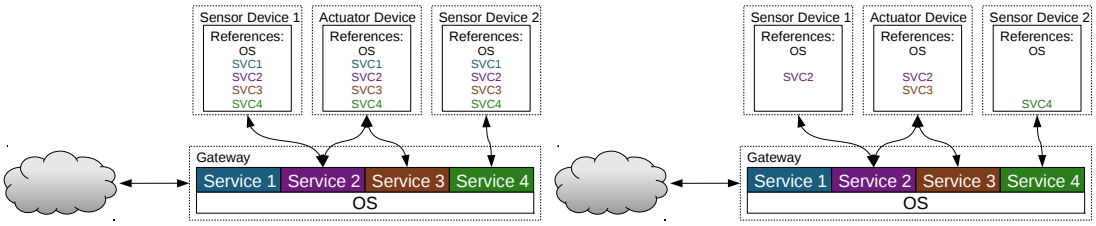


Figure 1. A common architecture: Many *Constrained Devices* communicate with a central device or gateway (*Full Device*). The reference configurations on each device is traditionally composed of binary measurements of all gateway modules (1a) although only some of them are important for this specific device (1b).

on the central device. Moreover, every time any of the services is updated, the reference configuration on all other devices has to be updated too. A superior solution has to reduce the complexity of the reference configuration to a minimum, as shown in Figure 1b. The challenger only has to know all services that influence the communication partner. For other services, the central station has to prove that there is no possibility for them to influence the challenger's services of interest.

Previous approaches based on measuring software-binaries [12] or security properties [13, 14] suffer from the problems of too many possible configurations or the requirement of a Trusted Third Party (TTP). Systems based on information flow analysis [15, 16] depend on comprehensive access control policies for all modules on the prover platform. Recently, the concept of privilege-based attestation has been proposed [17]. If a module does not have the privileges or permissions to harm the integrity of the targeted function on the prover, the challenger does not have to know a reference measurement. This approach could significantly reduce the complexity of the reference measurement list. However, until now no implementation of this scheme exists.

In this work, we provide the first usable design of this concept. We contribute a comprehensive trusted computing architecture, implemented on top of an Internet of Things (IoT) middleware. It combines binary-, property- and privilege-based measurements with a focus on a low overhead. In particular, this paper provides:

- The integration of a trusted computing architecture into IoTivity, an existing IoT middleware. To the best of our knowledge, this is the first comprehensive solution that brings remote attestation at system level to this domain.
- A transparent remote attestation protocol. Security is done under the application layer and high level services can focus on functionality.
- The application of different remote attestation methodologies in the IoT domain to reduce the set of known reference configurations. Therefore, compared to existing solutions, the approach is also practicable for systems with a high amount of services/devices.

Moreover, we created an experimental test environment to evaluate the architecture based on a virtual test-bed. Our solution provides simple methods for bootstrapping and configuring trusted relationships to enable authenticity for inter-device communication in ecosystems with device and vendor diversity. Furthermore, the system enables investigation of additional attestable properties for prospective devices and services. Similar to many IoT middleware implementations like IoTivity or AllJoyn, *Thingtegrity* distinguishes between *Full Devices* and *Constrained Devices*. In this work, we focus on the attestation of the integrity of *Full Devices* to *Constrained Devices* by reducing the size and dynamics of the configuration measurements. The attestation of *Constrained Devices* is out of the scope of this paper and left for future work.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the proposed system. Subsequently, Section 4 provides implementation details and explains how the architecture is integrated into IoTivity. Section 6 discusses the suitability of the approach based on an exemplar system that is introduced in Section 5. In Section 7, the benefits and the drawbacks of the system, as well as directions regarding our future work are summed up.

2 Background and Related Work

Trusted computing generally aims to build more secure systems by the implementation of different features. One of these features is remote attestation. This section describes the basic process of this concept and discusses existing methods that generate configuration measurements and verify them on the challenger.

2.1 Remote Attestation

Remote attestation is the process of proving the configuration of a system (prover) to another entity (challenger). In order to integrate this process, the prover has to provide a Root of Trust for Measurement (RTM) and a Root of Trust for Reporting (RTR). The RTM is in charge to measure properties that reflect the prover's system integrity (i.e., the integrity of all other software components on the system). Since malicious software would be able to change the taken measurements afterwards, a RTR is used to securely store this information and to protect it from malicious forging. Furthermore, the challenger has to comprise a policy or reference, that enables the verification whether the measured configuration represents a non-compromised system and a protocol for secure exchange of this information has to be in place.

Usually, the challenger sends a random value, called nonce, to request the prover's configuration. The prover signs its measurement (taken by the RTM), as well as the

nonce with its private key. Both the storage of the measurements and the signature is normally done by a dedicated hardware (the RTR) to prevent software from tampering with these values. One common way is to use a TPM [6] and perform the *TPM_QUOTE* operation. However, other technologies like ARM's TrustZone [18] or Intel's Trusted Execution Technology (TXT) [19] enable similar functionality. The challenger is now able to verify whether the retrieved measurement complies to its policy and check the signature with the public part of the prover's key in order to ensure data integrity. Both, TPM- and TrustZone-based attestation methodologies are too complex and expensive for many low-end embedded systems. Therefore, more lightweight approaches to enforce isolation of security-critical code have been introduced (e.g., [20], [21], [22]). These solutions enable attestation of tiny devices and would extend our system to also integrate mutual attestation for this class of devices.

2.2 Configuration Measurement and Verification

In order to attest the integrity of different devices to each other, the integrity of their configuration has to be measured. Basically, the configuration is represented by the software components running on the device. A variety of schemes and implementations that tackle this problem exist in the literature. Remote attestation methods for binaries, properties, security policies and platform-specific permission-systems have been introduced. However, the mapping of these concepts into the IoT domain is not a trivial matter due to resource and connectivity constraints.

The Integrity Measurement Architecture (IMA) [12] generates a measurement list of all binaries and configuration files loaded by the system. The cumulative measurement (i.e., hash) of the measurement list is extended into a Platform Configuration Register (PCR). To attest the system's state, the prover sends the measurement list to the challenger and proves its integrity with the help of the TPM. Binary measurement approaches are not suitable for systems with many different or dynamic configurations because each challenger has to maintain a comprehensive list of known 'good' configurations. Especially when system updates or backups are taken into account, the set of possible configurations may grow to an unmaintainable size. Moreover, the verification of all binaries is not necessary every time. The challenger might only be interested in modules which may affect the integrity of the target software. Our work uses IMA for the attestation of high-privileged software components.

Property-based attestation [13, 14] overcomes some issues of binary-based methods. A challenger is only interested in whether the prover fulfills particular security properties (e.g., strict isolation of processes). Therefore, a set of possible platform configurations is mapped to different properties. This approach eliminates the need for comprehensive lists of reference configurations on the challenger by the introduction of a TTP which is in charge of the mapping. Similar approaches focusing on privacy-preserving features [23] do not need a TTP and use ring-signatures to protect the prover's configuration from exposure. In this paper, we use this concept to sign reference measurements.

Another group of approaches use information flow analysis based on security policies [15, 16]. These approaches model all possible communications between processes. The basic idea is that a high-integrity process is successfully attested if all binary measurements are valid and there is no possible information flow from low-integrity to high-integrity processes. These approaches reduce the number of platform configurations since only a small set of system and high-integrity applications has to be measured. However, they rely on well-defined security policies and the generation of additional filter-components. In our work, we do not rely on existing policies or descriptions. They are generated at execution time.

Similar to policy-based and information flow based methods, PRIvilege-Based remote Attestation (PRIBA) [17] tries to reduce the information needed by the challenger by using privileges of software modules as trust properties. For software modules that have privileged access on the executing prover, binary measurement is used. All other modules are parsed for privileged calls to the system library to generate a privilege measurement of the module. The challenger is able to decide whether the measured module violates the prover's integrity by checking the measurement against a policy. The presented approach potentially reduces the size and the update frequency of the challenger's reference measurements. However, until now only the basic concept has been presented and no implementation exists. Neither the measurement of a modules's privileges nor the verification against the policy has been investigated. In our work we fill this gap by implementing privilege measurements as a central part of the trust properties used to attest a system's configuration.

3 System Architecture

This section provides an overview of *Thingtegrity*, our proposed trusted computing architecture and its underlying ideas. In Section 4, we will describe how the architecture is integrated into an IoT communication stack, namely the IoTivity middleware.

3.1 Overview

Thingtegrity aims to enable mutual verification of the integrity between these devices by introducing a transparent trusted computing architecture that enables remote attestation in this domain. To enable this attestation, the configurations of the devices have to be measured.

In this work, we focus on the configuration of *Full Devices*. These devices usually have a more complex and dynamic configuration, while their challengers are constrained. Therefore, the reduction of these measurements is an important first step to generate a trusted computing architecture in this domain. However, the architecture can be used for both type of devices and will be extended in future work.

3.1.1 Remote Attestation

Whenever two services on two different devices want to communicate they have to execute the following steps:

- Set up a secure connection: Before any communication, a secure connection that provides confidentiality, integrity and authenticity has to be set up. This is done with DTLS in the communication stack (IoTivity).

- Mutual attestation: Each service checks the measurement of the counterpart's configuration against the corresponding communication policy. This communication policy defines the rules the counterpart has to comply with to be trusted.
- Actual communication: If both services trust their communication partner, they can exchange information on the secure channel.

As mentioned in Section 2.1, additional hardware support (the RTR) is needed for storing and reporting these configurations. For simplicity a TPM is assumed to perform these actions in the remainder of this paper (although other hardware options like ARM's TrustZone are also possible).

3.1.2 Configuration Measurement and Verification

While the RTR enables storing and exchanging of the measurements in a tamper resistant way, components that are able to measure (prover) and to verify (challenger) the configurations are needed. As mentioned before, a variety of schemes exist for this challenge. *Thingtegrity* aims to use privilege-based attestation [17] to attest the integrity of the different services of a *Full Device*. These devices can contain many independent services. Therefore, this approach potentially minimizes the memory overhead for reference configurations, as well as the communication overhead. However, only the theoretical idea has been discussed for this privilege-based attestation. Hence, some technical implications have to be considered here: First, the privilege measurement unit requires 'measureable' accesses to privileged system functions. Therefore, we introduce an API with appropriate access granularity, which is discussed later. Furthermore, the system has to ensure, that these measured accesses are not circumvented at runtime. This is ensured by a sandbox. In order to enable a simple integration, we designed the introduced API in a way that enables automated generation of sandbox-policies at service-startup. The privilege-measurement unit is the RTM for this type of measurements. However, privilege measurements of this component as well as other low-level components cannot be taken. Therefore, we integrated the existing IMA [12] implementation for Linux into our framework to enable binary-measurements.

For verification, we introduce a simple policy that enables the decision whether the communication partner's integrity is intact. However, through the IMA-based measurements, the reference configuration lists may be too big and too dynamic to be handled in a network of constrained devices. Therefore, we also implemented a property-based attestation scheme, where measurement lists are signed by Trusted Third Parties (TPP). Additionally, we use the authentication mechanisms of the underlying communication protocol to integrate authentication of the device hardware.

3.2 Framework Architecture

The components of a *Full Device* are shown in Figure 2. Basically, a *Full Device* is composed of a hardware platform, a software platform and services. The hardware provides security features that enable any kind of remote attestation. The software platform consists of the operating system, the system libraries and the service framework.

The system libraries and framework provide functions to access the operating system and helper functions for common tasks. A service represents an actual application running on the platform. *Thingtegrity* distinguishes between privileged and non-privileged services. Privileged services have direct access to the system functions and thus comprehensive system access. Based on the underlying operating system, this access may be restricted through an access control system. Non-privileged services, however, do not have direct access to system resources. These services are initiated by the *Thingtegrity* runtime. The runtime generates a sandbox for each service that prohibits direct system access. Instead, the runtime provides an Inter Process Communication (IPC) interface that enables fine grained resource access to services. As described further on, this approach enables privilege measurement and ensures that services cannot access resources in an uncontrolled way. To enable configuration measurement, the operating system contains the measurement units. The binary measurement unit is in charge of taking binary measurements of all libraries and services while the privilege measurement unit generates privilege measurements of unprivileged services. Similar to IMA, the integrity of the measurement results are ensured with the help of a TPM.

Each device has to manage a platform identification key (pinned to the hardware) that is used to authenticate the hardware platform to other devices. As illustrated in Figure2, this key is stored in the TPM to protect it from software access. However, assuming a proper isolation by the operating system, the key can also be stored conventionally in the device's non-volatile memory.

Moreover, each device manages a list of Trusted Third Party (TTP) certificates and property signatures. A TTP property signature is used for property-based attestation and is a signed tuple of the software module's name, a hash (binary measurement) of its executable and the property name. Currently, *Thingtegrity* only uses one property, named *TrustedByThirdParty*. This property indicates, that a third party (e.g., the device vendor or the system administrator) trusts this binary and it is allowed to execute on the system. Each device manages the list of TTP certificates that contain the public part of their keys to verify the property signatures of other devices.

In contrast to *Full Devices*, a *Constrained Device* has a reduced feature set. They are simple devices like sensors or actuators that do not require support for highly dynamic services. Basically, their architecture is similar to Figure2. However, since there is no need for non-privileged services, they neither contain a *Thingtegrity* run-time nor a privilege measurement unit.

3.3 Measurement

In order to implement integrity assurance, the software components of the prover have to be measured. To achieve this, *Thingtegrity* uses binary measurement and privilege measurement. Here, the measure-before-execute paradigm is used: Prior to the execution of a new software module, a measurement of the module is taken and stored. Moreover, this measurement is extended to the PCR of the TPM. Since a PCR cannot be changed arbitrarily, malicious soft-

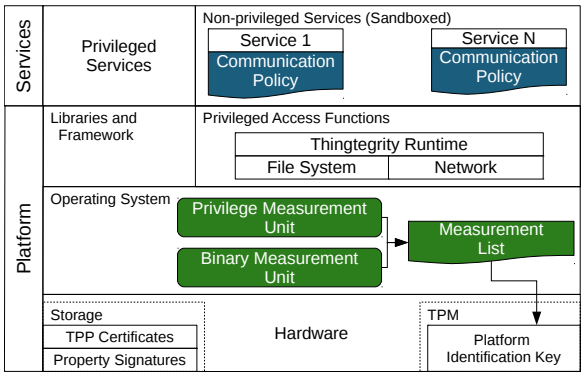


Figure 2. The components of a *Full Device*. All devices securely store a platform identification key that is used for authentication. Besides normal (privileged) services, the runtime confines non-privileged services that may or may not be able to influence each other.

ware is not able to deny its execution. *Thingtegrity* performs binary measurement of all services and additional privilege measurements of non-privileged services.

3.3.1 Binary Measurement

Binary measurements are taken at different tiers. To prove the integrity of the operating system, the boot process has to be measured. Therefore, at each boot stage, the next module loaded is hashed and extended into a PCR prior to its execution. This process is called an authenticated (or trusted) boot. The very first module is thus never measured (as there is no former module). Therefore, the first module should be as small as possible to reduce the attack surface and some measures should exists that prevent its substitution (e.g., write protected memory or hardware based solutions [24]). In the simplest architecture, a small bootloader initiates the TPM, measures the operating system kernel and executes it.

After this so called chain of trust is built up, the operating system is in charge of measuring the remaining modules. Here, *Thingtegrity* uses an extended version of IMA [12]. The modifications are primarily concerned with the measurement entry format required to fit into the global measurement list. A measurement entry of a binary measurement consists of the hash of the binary. This module also adds the measurements taken during the boot process (i.e. the measurement of the operating system) to the measurement list.

3.3.2 Privilege Measurement

The privileges of non-privileged services are measured to understand what kind of actions the service is able to perform. Whenever a non-privileged service is executed, the *Thingtegrity* framework generates a sandbox and initiates a privilege measurement for this module. The generation of these measurements is done by parsing the external symbols (i.e., function calls to system libraries) that access system resources from the service's executable. A found *fopen* call, for example, reveals that the service is able to access files on the system. However, this information is not very useful. A service that has read access to other service's files has completely different privileges compared to a service that only accesses private files. Since this information is provided by function parameters that are not static (i.e., they cannot be parsed from the executable), *Thingtegrity* provides a finer grained API to resources. As shown in Table 1, *Thingtegrity* currently provides functions for file and network access. In Section 5, we show that the chosen granularity provides enough information to enable privilege-based attestation for an exemplar system. Moreover, the API is coarse enough to make it feasible to use; it also allows the migration of legacy software with little effort.

Table 1. The fine grained API for resource access provided by the runtime.

Name	Type	Access Method
openPrivate	Open service-private file	
openGlobal	Open other service's file	Read, Write
openSystem	Open system-wide files	Read, Write
openTemp	Open files in temp-system	
createSocket	Create a network socket	Client, Server

The sandbox does not allow direct resource access for non-privileged modules. Whenever the service has to allocate a resource (like a file or a network socket), an IPC call is performed via the interface. The framework performs the actual allocation and forwards the resource descriptor to the service. With this sandbox, we ensure that the service cannot hide a resource access from the framework. As an example, the service may try to directly use low-level system calls in an obfuscated way. The privilege measurement unit may not be able to decode such calls and the access would not be measured. However, the sandbox prevents such calls on a lower layer and all resource accesses have to be made via the given API.

3.3.3 Measurement List

The measurement list is the container that a prover uses to store all its measurements. It is composed of a list of measurement entries. A measurement entry consists of the module name, the measurement type and a value. The entry-type is *binary* or *privilege*, depending on the measurement unit generating the entry. For binary measurements, the measurement value is the hash of the executable representing the module. A measurement entry of a privilege measurement contains a set of Resource Access Descriptions (RAD). RADs contain the resource type and additional attributes based on the type. In the current version, the resource type can only be one of *file* or *network*. A *file*-RAD contains the file type (equivalent to the API functions in Table 1) and the access type (read/write) as additional properties.

Table 2 illustrates an exemplary measurement list. The OS and the framework are measured with the binary measurement unit. Two services are running and both are measured with the binary and the privilege measurement unit. While *CalcService* only provides a calculation service on the network and thus does not need to access the disk, *StorageService* has access to its private files.

3.4 Integrity Assurance

The described building blocks enable a scalable trusted computing architecture for heterogeneous devices. If a *Full Device* has to prove its integrity to a challenger, various aspects have to be considered:

Table 2. An exemplary measurement list with different types of measurements.

Name	Type	Value
Platform OS	binary	hash=0cedac001ab4
Framework	binary	hash=b607c8734a9e
CalcService	binary	hash=1223bccdef66
CalcSerivce	privilege	RAD1={network}
StorageService	binary	hash=84fedacd2323
StorageService	privilege	RAD1={network} RAD2={file,Private,rw}

- A *Full Device* is a composition of different components from potentially different vendors. A component may be the hardware platform, the software platform (OS, framework) or a service.
- The overall system or the current cluster has an (probably human) owner or administrator that defines the policy that describes which devices and services are allowed in the system.
- Since smaller devices may be battery-powered and RF communication is expensive in terms of energy, the communication overhead for integrity assurance has to be minimal.
- A *Full Device* may integrate a variety of services the challenger is not interested in. If the challenger has to know all services the prover could possible execute, the system may not be feasible due to the high administrative overhead.

Thingtegrity combines the principles of binary attestation and privilege-based attestation to attest the integrity of the prover's state and introduces some security properties to reduce the overhead for communication and computation.

3.4.1 Attested Components

Table 3 lists the different components and the corresponding integrity assurance method based on the measurement technologies described above. Although there exists work about measuring the integrity of hardware in the literature (e.g., [25]), *Thingtegrity* does not take into account this aspect. However, the hardware is authenticated through the Platform Identification Key (PIK). The used keys are stored in a tamper resistant way and the attestation based on a TPM is somewhat secured against hardware attacks[1]. A challenger only communicates if the prover's public key is known as trusted. This is ensured with a digital signature during the initiation of the communication. Therefore, depending on the key distribution process, it is not possible to add a malicious node to the system.

Platform software consists of two parts. The operating system and the framework and all services are attested with an authenticated boot process and IMA (binary attestation. For non-privileged services, also a privilege-based attestation is used.

3.4.2 Integrity Assurance Process

Figure 3 describes the integrity assurance process. The prover tries to initiate the connection by sending a connection request, signed with its PIK, to the challenger. If the challenger is able to verify the PIK, a secure connection is established. This implies that the challenger has to know the prover's PIK prior to the communication. The challenger provides a nonce to prevent replay-attacks. Together with the PCR (i.e., the hardware-protected proof of the measurements), the nonce is signed by a TPM key to create the certificate for the measurement list.

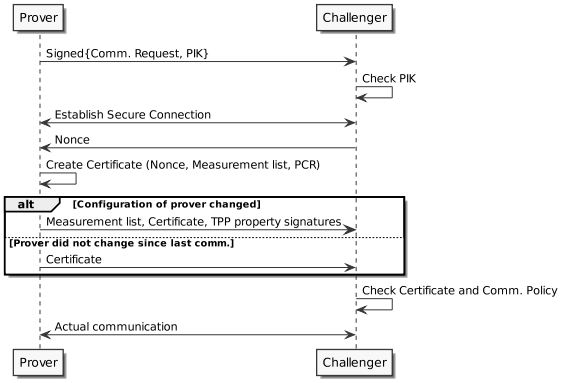


Figure 3. Attestation of a system configuration: After a secure connection is set up, the prover provides all data the challenger needs to verify its integrity.

With the measurement list, its certificate and possible property signatures, the challenger is able to verify the prover's integrity. The certificate ensures the integrity of the measurement list and the verification unit on the challenger checks whether the measurement list conforms to the communication policy. Since the measurement list and the property signatures sometimes imply a high communication overhead and only change in the relatively rare cases of software updates on the prover, the challenger can cache them and only request a fresh certificate on further communications.

3.5 Verification

For verification, we use a very similar concept as proposed for privilege-based attestation [17]. In contrast to usual binary attestation, privilege-based attestation only considers the service which is used by the challenger and its dependencies. For all other services, it is ensured that they are not able to manipulate the integrity of the targeted service(s). This is accomplished by executing the rules defined in the communication policy and comparing the reference measurements to the prover's measurement list based on these rules.

3.5.1 Communication Policy

The communication policy proposed for privilege-based attestation is very complex since it offers a very high flexibility that enables comprehensive information flow analysis for all services. Since this is a task that may be to heavyweight for constrained devices, *Thingtegrity* currently uses a very simple policy which is forced for all modules. The policy simply states that no other service is allowed to access a service's (or one of its dependencies) private files in read or write mode. Although this policy limits the flexibility of the system, it is a good representation of the generic 'no other

[1] TPM 1.x chips are considered broken for physical access [26]. However, future revisions or on-chip solutions may reduce this attack surface

Table 3. The measured components and the corresponding integrity assurance method.

Component	Assurance Method	Technology
Hardware	Authentication (Platform Identity Key)	Digital Signature
OS	Binary Attestation	Authenticated Boot
Framework	Binary Attestation	IMA
Privileged Services	Binary Attestation	IMA
Non-Privileged Services	Binary Attestation, Privilege-Based Attestation	IMA/Privilege Measurement

service is able to influence the service's integrity' policy and feasible enough to show the functionality of the prototype.

3.5.2 Verification Unit

The reduced communication policy enables a very lightweight verification unit. All modules from the prover's measurement list are separated into a privileged and a non-privileged list. The privileged list contains the following entries:

- Modules where no privilege measurement exists (OS, framework, system libraries, privileged services).
- The remote service that is targeted by the challenger (communication partner). This information is either provided by the challenger's endpoint service (as in our implementation) or by the prover's *Thingtegrity* framework.
- All dependencies of the communication partner. This information is provided by the prover's *Thingtegrity* framework since the challenger may not be aware of these relations.

All non-privileged services are checked as to whether they comply with the communication policy. If a service violates the policy, it is added to the privileged list. It is thus considered as dependency (from the security point of view) and verified by binary attestation. For all privileged services, the binary measurement must either be in the local reference measurement list or certified by a TTP property signature (i.e., the prover provided the signature and the corresponding TTP certificate is in the local list).

4 IoTivity Integration

We integrated the architecture described above into IoTivity, a framework for IoT applications. For the current version, we targeted *Full Devices* with Linux on ARM and x86 platforms. However, we are working on integrating support for *Constrained Devices* on platforms like Arduino.

IoTivity is a resource-based, RESTful framework that provides device and resource management, as well as a unified communication stack for IoT. It defines devices, resources and operations. A device provides resources to the outer world. A resource is a component that can be viewed or controlled by another device. An example of a resource may be a temperature sensor or a light controller. Moreover, IoTivity offers resource topologies and virtual resources. Via a RESTful API, IoTivity supports different operations (e.g., GET and PUT) on these resources. Based on these components, IoTivity provides functionality to register a resource, find a resource in the network and to perform operations on remote resources. For secure connections, we use IoTivity with DTLS based on Elliptic Curve Cryptography (ECC). To reduce the overhead, these authentication keys are currently used as the PIK.

4.1 Thingtegrity Runtime

As mentioned before, *Thingtegrity* consists of the runtime that sandboxes non-privileged services, the measurement units and the remote attestation process on top of a secure channel. In our implementation, we use many components that are already implemented in Linux and IoTivity to keep the overhead minimal.

4.1.1 Sandboxing and Resource Interface

The *Thingtegrity* runtime is the central point that manages all non-privileged services. It is in charge of their execution, sandboxing and resource access and achieves this with the following parts:

- A service that allows deployment or update of other services on the system to authorized users.
- A chroot jail for all non-privileged services.
- One local socket for each service to enable communication with the runtime.
- An interface that provides access to the system resources via the runtime and replaces libc's functions.

The deployment of a new service or the update of an existing service is done via the *Thingtegrity* deployment service. This is a privileged service that adds (or removes) services to the runtime. Currently, this is authenticated with a simple password-check. *Thingtegrity* generates a directory structure for each service. This sandbox contains the executable, the libraries used by the service and the local socket file. Prior to the execution, the runtime chroots into the directory to prevent the service from accessing the file system directly. Generally, chroot jails are not a security feature and fail if the guest applications gain root access in their confined environment. However, in combination with *Grsecurity* that mitigates many of chroot's security problems, this type of sandboxing is suitable for our prototype since we execute the services with a very restricted user. Another advantage of this approach is that the runtime has control over the libraries used by the service. Currently we only provide IoTivity, the *Thingtegrity* interface and their dependencies. To simplify the development of new services, we also added a modified version of *Qt*, which uses the *Thingtegrity* interface for file access. While such libraries may not be feasible on a *Constrained Device*, a *Full Device* should normally have enough resources to allow their usage.

Analogous to the sandboxes, a private file directory for each service is generated to store their private data. In order to access a file outside the chroot jail, an application uses the *Thingtegrity* interface that provides resource allocation functions similar to the interface introduced in Table 1 with dedicated functions for the read and write variations. The framework checks whether the function call is valid (i.e.,

this type of resource access was measured before for this service), opens the file and passes it back to the service.

4.1.2 Integrity Measurement

Currently, *Thingtegrity* uses Linux's IMA implementation as the binary measurement unit. It is configured to measure all files that are executed. Privilege measurement is done by the *Thingtegrity* framework in user-space. The runtime extends *GNU nm* to read the external symbols of the executables. These symbols are mapped to resource access descriptors and added as privilege measurements to a dedicated privilege measurement list. Both measurement lists use a different PCR. However, when a challenger requests the lists, they are merged into a combined structure.

4.2 IoTivity Extensions

IoTivity differentiates between secure and non-secure resources. Based on this property, a secure connection is used. We added a property, called *RequiresAttest*, that indicates, that a resource also needs a trusted client to enable communication. Since our attestation method relies on a secure connection, this property implies the secure resource property.

Whenever a device wants to enumerate IoTivity resources in the network, it multicasts a request to all other devices that are providing resources (server). Each server responds with a list of all its resources and their properties. If attestation is required for one of the listed resources, the server also adds a nonce as a header option to the response. Based on these properties, the client decides whether it has to attest its integrity to access a resource. In this case, the client adds a header option to the GET or PUT request, and adds the attestation information to the payload. As mentioned before, this information is either the complete measurement list, the certificate and the property signatures, or the certificate only (in case the server already has a cached copy of the current measurement list). We extended IoTivity to extract this information from the payload and forward it to the runtime that checks the response. Therefore, this process is transparent to the actual service running on top of the framework.

5 Use Case

In order to evaluate the trusted computing architecture, we generated a set of IoTivity services that simulate an exemplary home automation use-case where products of different vendors are used in one system that is controlled by its owner. Although this use-case is relatively simple, it represents all basic mechanisms and since *Thingtegrity* is non-intrusive and transparent to the actual system, the results in other environments such as an industrial automation system or a cluster of a bigger network would be similar. Figure 4 illustrates the evaluated system. The Control Center (CC) is a *Full Device* running on an ARM single board computer and hosts a number of services. We also simulated a set of *Constrained Devices*, each in a virtual machine running on an off-the-shelf PC. The *Constrained Devices* represent simple sensors or actuators. Since we are not interested in functionality here, they either provide or consume some random values. We assume that the devices and services are from different vendors. Therefore, they don't know or trust each other when the system is set up. However, some of them provide a known API so other services are able to request their data. In this evaluation, we assume that corresponding services and devices (i.e., temperature control and temperature sensors) are from the same vendor. Moreover, a user (the owner) exists, who is interested in retaining control of the overall system. The services running on the CC provide the following functionality:

Deployment: This service is used to deploy all other services on the CC. As described in the previous section, this is a privileged service and part of the framework. The CC and the framework is delivered by $Vendor1(V1)$

Bootstrap: This service is also part of the framework and used to bootstrap the trusted relations between the CC and the *Constrained Devices*.

Backup: In this scenario, the system owner created a backup service that collects and stores private data from all other services. Therefore, this service has global file access.

TemperatureControl: This service periodically reads data from different temperature sensors and writes values to actuators. The service and devices are delivered by $Vendor2(V2)$.

AccessControl: This service represents an access control system to an apartment or house. An authenticated *Constrained Device* (e.g., a smartphone) is able to control the actuator (e.g., the door lock). Moreover, the service maintains a 'presence' state of the owner. The user is able to inform the service about its absence. This information can be requested by other authenticated services. The service and devices are delivered by $Vendor3(V3)$.

LightControl: This service represents a light control service that allows authenticated devices (like switches or smartphones) to control the state of light bulbs. Moreover, this service requests the 'presence' state from the *AccessControl* service and sporadically switches lights on or off in case the absence lasts longer than a defined time interval T. The service and devices are delivered by $Vendor4(V4)$.

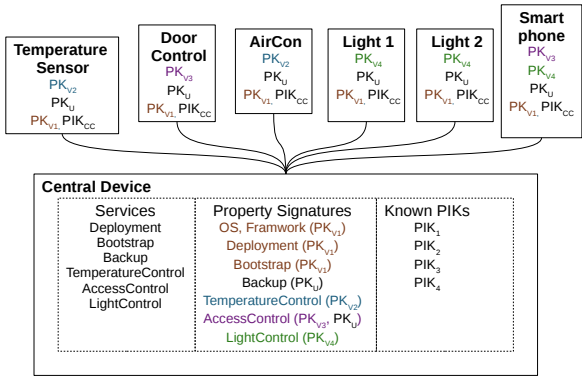


Figure 4. The exemplary system for investigation of the implemented architecture. The Control Center (CC) carries different applications from different vendors that communicate with different types of constrained devices.

The implementation of the services revealed by the privileges and dependencies are shown in Table 4. All services, except *Backup* only access private data. Moreover, *LightControl* depends on the integrity of *AccessControl*, because this service is indirectly able to manipulate the actuators'

Table 4. Control Center (CC) services with their file privileges (Private, Global, Read, Write) and dependencies.

	Name	Priv.	Dependency
1	Deployment	-	
2	Bootstrap Service	-	
3	Backup	G(R)	
4	TemperatureControl	P(RW)	
5	AccessControl	P(RW)	
6	LightControl	P(RW)	5

states.

5.1 Bootstrapping Trust

In order to set up the system, all devices have to be configured with some basic information like the PIK or TTP signatures. To create a feasible trust-architecture, this process should be as lean as possible. In our scenario, we have some basic assumptions:

- Every device is provisioned with a PIK in the manufacturing process.
- Each vendor defines itself as TTP. Each device therefore has the public key of its vendor in its TTP list.
- A vendor that releases a service as a binary, also provides the corresponding signature of the *TrustedThirdParty* property. This is also true for the platform vendor, who signs the OS and framework measurements and deploys these signatures with the CC. In the current implementation, only a plain signature of the binary's hash is generated since there are no other properties.
- The user trusts the vendor of the CC-platform, since this device is always able to access the user's data.

A PIK exists for each device $(PIK_{CC}, PIK_1 ... PIK_N)$ and a property signature key for each vendor $(PK_{V_1} ... PK_1)$ as well as for the user (PK_U). Each of these keys consist of a public and a private part.

Based on these assumptions, the user is able to create the trusted relationships between all its devices with few steps:

- Get ownership of one device (CC) and set it up (install software).
- If there are dependencies, configure them.
- For each other device: Add it to the network (which eventually needs a manual confirmation).

First, the *Bootstrap* service is used to set the user as the device owner by loading the public part of the user's PK_U to the CC. This action is only possible once and secured by a one-time password (e.g., printed on the device). Subsequently, the user deploys all devices and property signatures via the deployment service. Moreover, the user configures the *LightControl* service to use the *AccessControl*'s presence feature. PK_U is used to sign a *TrustedByThirdParty* property for the *AccessControl* service. This indicates that this service is trusted in the user's system. This signature is stored to CC with the *Deployment* service.

Whenever a *Constrained Device* joins the network, it tries to locate a *Bootstrap* service. If the communication succeeds, the service automatically deploys the property signature keys of the user and CC's platform vendor to the *Constrained Device*. Currently we use the trust on first sight paradigm. A sensor or actuator device therefore only trusts the first bootstrap service it is able to find. However, we could also use other mechanisms like the one-time password or Password Authenticated Key Exchange (PAKE) (e.g., [27]), if the *Constrained Device* is a more complex device and has some kind of key pad). Another possibility would be one-time passwords that are printed onto the device. In order to allow the device to access the network, an authenticated user has to confirm its membership via the *Bootstrap* service. Technically this adds the device's PIK to the CC's known and trusted PIK list.

In our test system, the smartphone is modeled as a *Constrained Device* that executes two services (for communication with the *AccessControl* and the *LightService*) on the CC. Therefore, the property signature keys of both vendors are known by this device. In a real-world scenario the smartphone should be modeled as *Full Device* with another vendor that is trusted by the CC.

5.2 Integrity Assurance

After the initial configuration, devices which have to communicate are able to attest their integrity to each other. All devices are authenticated with their PIK. A sensor or actuator node is able to check the integrity of the CC by verifying the property signatures of CC's OS and framework against PK_{V1}. Moreover, these devices are able to verify the integrity of its corresponding service or its dependencies by using its vendor's or the owner's signature key.

Although the test system currently does not perform this task, the CC would be able to check the integrity of the other devices by verifying the property signature of their binary measurements. Since *Constrained Devices* do not contain the full framework, they only provide binary attestation. Moreover, all *Constrained Devices* that have to communicate with each other (e.g., the light actuators *Light1* and *Light2*) are able to attest their integrity to each other, because they share the same vendor key. The current implementation, however, relies on a TPM, what is not feasible for tiny devices. Therefore, more lightweight approaches (e.g., [20], [21], [22]) have to be included in the framework in order to enable mutual attestation in the future, as discussed in Section 2.

Whenever a vendor updates one of its services, the owner just has to deploy the update over the deployment service (or allow the vendor to push updates). The new property signature is automatically propagated to all devices and nothing has to be reconfigured. If no property signatures are used and the other devices manage reference measurements, the new reference only has to be pushed to devices that actually used the new service.

6 Evaluation

In order to build a feasible trusted computing middleware for IoT, the system has to provide an attestation mechanism that targets common threats in this domain without adding too much overhead in terms of administration, communication and computing to ensure scalability. With the help of the exemplar system described above, we are able to analyze the proposed system regarding common attacks as well as the additional complexity.

6.1 Security Evaluation

We analyze different methods of potential malicious modifications of the overall system and how they are detected. Moreover, we analyze the communication protocol and the current set of attested properties for future directions.

6.1.1 Attacker Model

Given that the sensed information or actuated environment of the device owner should be protected, this stakeholder is considered trusted in this evaluation. Since the devices may be exposed in a publicy accessible environemt, an adversary may have limited physical access to existing hardware (e.g., J-TAG access). Therefore, she has the possibility to modify the software configuration. Additionally, an adversary is able to modify existing hardware and may try to add new devices to the system. However, the adversary is not able to read or modify information, that is protected by additional hardware measures (e.g., the PIK).

6.1.2 System Modifications

Table 5 shows potential attacks on the system and whether *Thingtegrity* is able to detect the attack or mitigate the threat and what type of attacks have to be countered with other technologies. Basically, the system can be modified by manipulation of an existing or insertion of a new hardware or software module.

Hardware Manipulation: Adversaries with physical access to the system may modify existing hardware. They might be able to forge sensor values or directly read/write unprotected electrical signals. Other devices would not be aware of this security breach because the altered device would authenticate with its PIK and the software is not modified. However, this type of attack usually demands on indeep system knowledge and high effort. If potential motivated (in terms of revenue) attackers have physical access to the devices and their direct environment, other measures like physical protection or plausibility checks of signals have to be in place.

Static Software Manipulation of a Privileged Service; Here, the term static manipulation means that the binary of a module is changed statically (i.e., persistent). A modified privileged service or a modification of other system components would cause another binary measurement that is not known or issued by any other entity. This measurement would violate the communication policy of other devices and therefore they would refuse to communicate. Adversaries are thus isolated and cannot access the network. However, they may be able to perform Denial of Service (DoS) or jamming attacks on the physical layer to reduce availability of other services.

Static Software Manipulation of a Non-Privileged Service: If a non-privileged module is altered in a way that changes its properties (privileges), these changes are reflected in the privilege measurement. Therefore, this case is comparable to manipulation of a privileged service. If the non-privileged service is changed without escalating the properties, the integrity of other modules is not harmed in case of a proper communication policy. Here, this means that the module is still not able to access another modules files. As discussed later, this policy may not be sufficient for all possible systems and security properties (e.g., availability). A device that is directly communicating with this (maliciously modified) service considers it as privileged service, and therefore is able to detect the manipulation. Again, an adversary is not able to maliciously modify the system without detection.

Dynamic Software Manipulation of a Privileged Service: Additionally to static manipulation, we have to consider runtime code modifications. Examples may be buffer overflow attacks or Return Oriented Programming (RoP) attacks. Since the binary measurement is taken prior to the execution, these modifications are not reflected in the measurement list and cannot be detected. Therefore, other mitigation techniques like a shadow stack (for example [28]), have to be used against this type of attacks.

Dynamic Software Manipulation of a Non-Privileged Service: In contrast to privileged services, a sandbox is generated based on the identified privilege measurement of the service. Therefore, similar to static manipulation, the service is at least not able to harm other service's integrity. However, it may perform malicious actions that comply with the services' sandbox. Therefore, the principle of least privilege should be enforced during service development.

Insertion of Hardware/Devices: Assuming that an adversary has no access to valid PIKs, additional devices are ignored by the system. However, again DoS or jamming attacks have to be considered.

Insertion of new Software Modules: Technically, the insertion of new software modules is the the same as statically changing a module. Therefore, the same considerations apply here.

6.1.3 Communication Protocol

The underlying secure communication protocol prevents message modifications on the channel. Moreover, the current *PSK* scheme pins a PIK to a device and therefore prevents man-in-the-middle attacks. As described above, it is not possible to (maliciously) add new devices into the system. This is an additional counter-measurement against this type of attacks. Moreover, the protocol is resistant against replay-attacks because of the used nonce. Therefore, the protection of the PIK, the key-exchange, as well as the quality of the random nonce should get special attention when implementing the system.

6.1.4 Measured Properties

Currently, we only measure binaries and privileges of services. As mentioned before, these properties may not be enough for security requirements of many real systems. Binary attestation does not protect against dynamic code changes and the overall measurements do not protect against a variety of attacks: For example a malicious service would be able to consume a high percentage of a hardware resource to prevent other services on the same device from working properly. Another possible class of attacks may be side-channel attacks. In future work, we will use the exemplary system described above to examine possible other properties.

6.2 Overhead

To evaluate the overhead of *Thingtegrity* in terms of communication, computation and memory, we compared the full

Table 5. Overview of possible attack types regarding system modifications.

Name	Description	Mitigation
Manipulation		
Hardware	Modification of the hardware of a device	x
Privileged (static)	Static modification of a privileged software module	✓
Non-Privileged (static)	Static modification of a non-privileged software module	✓
Privileged (dynamic)	Runtime modification of a privileged software module	x
Non-Privileged (dynamic)	Runtime modification of a non-privileged software module	(✓)
Insertion		
Hardware	Insertion of a new device	✓
Privileged	Insertion of a privileged software module	✓
Non-Privileged	Insertion of a non-privileged software module	✓

implementation with binary-attestation only (IMA). Figure 5 shows the results for our exemplar system. We used property signatures for binary measurements in both cases to make the results more comparable. The use of plain reference measurements would lead to comparable results, because analogous to the TTP keys, each device would have to manage keys to verify updates of the reference measurements. The actual memory and time complexity of the overhead highly depends on the used cryptograhic schemes (e.g., key size and signature verification complexity).

As shown in Figure 5a, the information stored on the constrained devices is singificantly smaller if *Thingtegrity* is used. Only TTP certificates for privileged services, the target service and its dependencies have to be in stored. Moreover, less stored keys are also reflected in a simpler bootstrap-process (less keys have to be provisioned), as well as in better scalability (if a new unprivileged service or device is added to the system, existing devices do not need an additional TTP certificate, as shown in Figure 5b). Moreover, a high number of TTP certificates increases the chance of one leaked private key that is considered trusted by the whole system.

However, this architecture increases the size of the measurement list and adds overhead from the additional privilege measurements, their verification and the sandbox. The extended measurement list with privilege measurements and dependencies increases the number of bytes that have to be transmitted on the network interface. Since many of the targeted applications are battery-powered with wireless network interfaces, this is a critical part. Therefore, we compared the sizes of the measurement lists for our test system with and without privilege measurements and dependencies. While the size of the binary measurement list is $429B$, the full list requires $506B$, an increase of 18.8%. Compared to the gain of information, the increase is relatively small. The measurement list also only contains four non-service entries, because we were able to merge some binary measurements for different libraries contained within the framework. Moreover, this overhead vanishes after the first communication because the measurement list is cached by the challenger. Additionally, it has to be stated that we do not send more packages than without attestation, since we integrate all information into the existing communication.

The process of measuring the privileges of software modules does not significantly affect performance. Since we only parse the headers of the executables, the time used for measurement is similar to hashing to whole binary [17].

Regarding the fine grained resource access API, two aspects have to be considered. First the IPC calls have an impact on performance. However, since resource access calls are normally relatively slow anyway, our measurements showed that the time overhead for *fopen* is only about $0.1ms$ (Off-the-shelf PC with SSD, cleared file system caches). Moreover, after the first access (i.e., the file is opened), the normal system API (like *read* or *write*) can be used. Although the fine-grained API is unfamiliar and we have not yet done usability studies, we believe the impact on service development is not significant. For the test system, we added wrapper functions for *libc*, as well as for the Qt framework (with different versions of *QFile*). Based on these library extensions and static analysis, we were also able to update legacy software to the new API relatively simply.

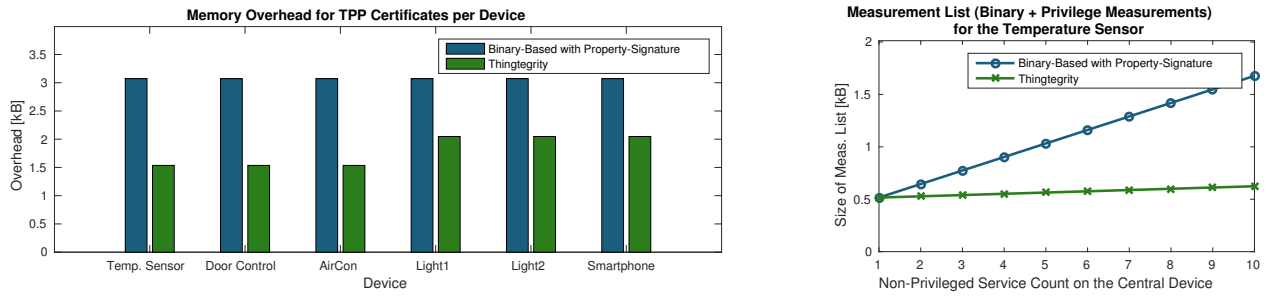
Similar results could be achieved by directly using sandbox policies or model carrying code instead of measuring the executables before their execution. Some of these methods are described in the related work. However, with the measure-approach, neither the module developer nor the system administrator has to generate these policies. Moreover, the system itself is able to decide what type of privileged functions are relevant. Therefore, updates of the communication policy model do not require an updated service.

7 Conclusion and Future Work

In this work *Thingtegrity*, a trusted computing architecture for systems with many devices that are constrained in terms of energy, connectivity and performance has been presented. We combined concepts from binary-, property- and privilege- remote attestation and integrated it into IoTivity. The architecture is transparent and hides the complexity of remote attestation from the overlying application. Additionally, we provide a testbed that enables the investigation of further attestable properties for future devices and systems.

As a first step, we implemented the system for the attestation of software configurations on *Full Devices*. We showed that the architecture enables a simplified bootstrapping of trusted environments. Compared to traditional remote attestation systems, the maintainability and scalability of the trusted relations is improved. This is achieved by reducing the complexity of configuration measurements. This reduces the memory and communication overhead significantly for systems with a high number of services or devices.

The next step is the integration of attestation of *Constrained Devices*. In order to enable support for real-world tiny devices, more investigation with regards to security architectures at device level, as well as the reduction of asymmetric cryptography has to be conducted. We thus have to provide support for other, non-TPM RTRs for hardware-



(a) (b)

Figure 5. The additional memory used to store TPP public keys (5a) per device, as well as the size of the measurement list that has to be verified on the temperature sensor (5b).

based signatures.

Based on the test system, it would be desirable to build more use-cases for different domains to explore the usability of the current resource access API. Moreover, support for other resource types such as sensors has to be added. The current sandboxing solution should also be replaced with a less intrusive method. Especially, regarding ports to other environments, this part should be interchangeable.

Based on future investigations, the communication policies should be refined. Currently only access to files of other services is considered. However, also other resources and IPC mechanisms have to be examined. Moreover, information flows are not the only threat to a service's integrity. A malicious module without any permissions may consume a lot of CPU or storage to prevent other modules from working correctly. Therefore, further properties should be introduced to prove attributes like computing capacity.

In summary, we showed that remote attestation is in fact feasible for IoT architectures and with the spread of common standards systems that are comprised of a high number of modules from different vendors are also capable of proving their integrity.

8 Acknowledgements

Thanks to Matthias Kovatsch for improving this paper by giving helpful comments during the shepherding process.

9 References

[1] Gartner Inc., "Analysts to Explore the Disruptive Impact of IoT on Business," in *Gartner Symposium/ITxpo*, 2014.

[2] BBC, "Not in front of the telly: Warning over 'listening' TV," 2015. [Online]. Available: http://bbc.com/news/technology-31296188

[3] A. Chapman, "Hacking into Internet Connected Light Bulbs," 2014. [Online]. Available: http://www.contextis.com/resources/blog/hacking-internet-connected-light-bulbs/

[4] D. Halperin, S. S. Clark, and K. Fu, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," *Proceedings - IEEE Symposium on Security and Privacy*, 2008.

[5] B. Miller and D. Rowe, "A survey SCADA of and critical infrastructure incidents," *Annual Conference on Research in Information Technology*, p. 51, 2012.

[6] Trusted Computing Group, "TPM Main Specificication Level 2 Version 1.2," 2006.

[7] S. W. Smith, "Outbound authentication for programmable secure coprocessors," *International Journal of Information Security*, vol. 3, no. 1, pp. 28–41, May 2004.

[8] M. Nauman, S. Khan, X. Zhang, and J. Seifert, "Beyond kernel-level integrity measurement: enabling remote attestation for the android platform," *Trust and Trustworthy Computing*, pp. 1–15, 2010.

[9] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to Remote Attestation," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pp. 1–6, 2014.

[10] R. Akram, K. Markantonakis, and K. Mayes, "Remote Attestation Mechanism based on Physical Unclonable Functions," *Workshop on RFID and IoT Security*, 2013.

[11] M. LeMay and C. a. Gunter, "Cumulative Attestation Kernels for Embedded Systems," *IEEE Transactions on Smart Grid*, vol. 3, no. 2, pp. 744–760, Jun. 2012.

[12] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *USENIX Security Symposium*, 2004.

[13] A. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms," *Proceedings of the 2004 workshop on New Security Paradigms*, pp. 67–77, 2004.

[14] M. Ceccato, Y. Ofek, and P. Tonella, "A Protocol for Property-Based Attestation," *Theory and Practice of Computer Science*, p. 7, 2008.

[15] T. Jaeger, R. Sailer, and U. Shankar, "Policy-Reduced Integrity Measurement Architecture," in *Symposium on Access Control Models and Technologies*, 2006.

[16] W. Xu, X. Zhang, and H. Hu, "Remote attestation with domain-based integrity model and policy analysis," *Dependable and Secure Computing*, vol. 9, no. 3, pp. 429–442, 2012.

[17] T. Rauter, A. Höller, N. Kajtazovic, and C. Kreiner, "Privilege-Based Remote Attestation: Towards Integrity Assurance for Lightweight Clients," in *Workshop on IoT Privacy, Trust, and Security*, 2015.

[18] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.

[19] James Greene, "Intel Trusted Execution Technology," *Intel Whitepaper*, 2003.

[20] D. Perito, G. Tsudik, and K. E. Defrawy, "SMART : Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," *Security*, 2012.

[21] P. Koeberl, S. Schulz, A.-r. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[22] F. Brasser, B. E. Mahjoub, A.-r. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny Trust Anchor for Tiny Devices," in *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[23] L. Chen, H. Löhr, M. Manulis, and A. Sadeghi, "Property-based attestation without a trusted third party," *Information Security*, 2008.

[24] J. Li, H. Zhang, and B. Zhao, "Research of reliable trusted boot in embedded systems," in *Computer Science/Network Technology*, 2011.

[25] C. Yu and M. T. Yuan, "Integrity measurement of hardware based on TPM," *International Conference on Computer Science and Information Technology*, vol. 3, pp. 507–510, 2010.

[26] E. R. Sparks, "A Security Assessment of Trusted Platform Modules," Tech. Rep., 2007.

[27] F. Hao and P. Y. a. Ryan, "Password authenticated key exchange by juggling," *Lecture Notes in Computer Science*, 2008.

[28] L. Davi, A. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," *ASIACCS*, pp. 1–22, 2011.