ISO/IEC TS 30168

Edition 1.0 2024-05

TECHNICAL SPECIFICATION

colour inside

Internet of Things (IoT) – Generic trust anchor application programming interface for industrial IoT devices

ISO/IEC TS 30168:2024-05(en)

ISO/IEC TS 30168

Edition 1.0 2024-05

TECHNICAL SPECIFICATION

colour inside

Internet of Things (IoT) – Generic trust anchor application programming interface for industrial IoT devices

INTERNATIONAL ELECTROTECHNICAL COMMISSION

ICS 35.020

ISBN 978-2-8322-8518-3

CONTENTS

INTERNET OF THINGS (IoT) – GENERIC TRUST ANCHOR APPLICATION PROGRAMMING INTERFACE FOR INDUSTRIAL IoT DEVICES

FOREWORD

1) ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.
2) The formal decisions or agreements of IEC and ISO on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC and ISO National bodies.
3) IEC and ISO documents have the form of recommendations for international use and are accepted by IEC and ISO National bodies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC and ISO documents is accurate, IEC and ISO cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC and ISO National bodies undertake to apply IEC and ISO documents transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC and ISO document and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC and ISO do not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC and ISO marks of conformity. IEC and ISO are not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this document.
7) No liability shall attach to IEC and ISO or their directors, employees, servants or agents including individual experts and members of its technical committees and IEC and ISO National bodies for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this ISO/IEC document or any other IEC and ISO documents.
8) Attention is drawn to the Normative references cited in this document. Use of the referenced publications is indispensable for the correct application of this document.
9) IEC and ISO draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). IEC and ISO take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, IEC and ISO had not received notice of (a) patent(s), which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at https://patents.iec.ch and www.iso.org/patents. IEC and ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TS 30168 has been prepared by subcommittee 41: Internet of Things and Digital Twin, of IEC technical committee JTC 1: Information technology. It is a Technical Specification.

This document contains attached files in the form of GTA API C header files and a secure element provider template that are cited in Annex A and Clause G.6. These files are intended to be used as a complement and do not form an integral part of the publication.

The text of this Technical Specification is based on the following documents:

Draft	Report on voting
JTC1-SC41/388/DTS	JTC1-SC41/413/RVDTS

Full information on the voting for its approval can be found in the report on voting indicated in the above table.

The language used for the development of this Technical Specification is English.

IMPORTANT – The "colour inside" logo on the cover page of this document indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

INTRODUCTION

Industrial Internet of Things (IIoT) devices face increasing security requirements. This insight is especially important as more and more devices become connected directly or indirectly to the Internet. It is essential that IIoT devices are prepared to perform secure communications for service interaction, monitoring, and control.

However, security is often still observed as rather complex by implementers and integrators. This perception often results in realization obstacles when the integration and use of security mechanisms and secure elements (SE) is wanted.

This document provides a versatile application programming interface (API) for security to allow a generic integration of SEs into IIoT devices. The API is vendor independent and also independent regarding the SE technology being deployed. This approach simplifies redesign for different SEs and supports software-hardware co-design for security. SEs offering different security properties facilitate the selection of an SE according to the intended use, protection goals, and other boundary conditions. The API aims at achieving high-level abstraction profiles for security services and mechanisms to avoid typical low-level interoperability complexity and implementation failures. Requirements and architectural constraints from IIoT applications shape the final design of the API and its usability.

The resulting API facilitates the security-by-design defined integration of security components within IIoT components on a large scale. The time-to-market for secured devices is accelerated. Stakeholders will benefit from higher security levels being available for lower prices. Application of updates and continuous improvements of security along the lifecycle of products and systems are facilitated.

The following stakeholders and their corresponding interests play a role for the generic trust anchor application programming interface (GTA API) definition:

- Manufacturers and users of industrial equipment

  Scalable use of adequate (hardware-based) security technologies depending on required security, multivendor support, migration strategy, or long-term suitability.

- Software developers

  Increased robustness due to use of a unified API.
  Ease of use for developers without dedicated security expertise.

- Manufacturers of security ICs or ICs offering security functions

  Promote use of hardware-based trust anchor technologies for IIoT devices.

- Conformity Assessment Bodies

INTERNET OF THINGS (IoT) – GENERIC TRUST ANCHOR APPLICATION PROGRAMMING INTERFACE FOR INDUSTRIAL IoT DEVICES

1 Scope

This document specifies a generic application programming interface (API) for the integration of SEs within Industrial Internet of Things (IIoT) devices. It considers needs from industrial usage scenarios and applications. This document also provides guidance for implementation, testing, and conformity validation.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEEE Std 802-2014, *IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture*

IETF RFC 1035, P. Mockapetris, "Domain names – implementation and specification", November 1987, available at https://www.rfc-editor.org/info/rfc1035 [viewed 2023-08-29]

IETF RFC 1779, S. Kille, "A String Representation of Distinguished Names", March 1995, available at https://www.rfc-editor.org/info/rfc1779 [viewed 2023-08-29]

IETF RFC 4122, P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", July 2005, available at https://www.rfc-editor.org/info/rfc4122 [viewed 2023-08-29]

IETF RFC 8446, E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", August 2018, available at https://www.rfc-editor.org/info/rfc8446 [viewed 2023-08-29]

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

- IEC Electropedia: available at https://www.electropedia.org/
- ISO Online browsing platform: available at https://www.iso.org/obp

3.1
attack
attempt to gain access to an information processing system in order to produce damage

Note 1 to entry: The damage can be, for example, destruction, disclosure, alteration, unauthorized use.

[SOURCE: IEC 60050-171:2019, 171-08-12]

3.2
generic trust anchor application programming interface
GTA API
set of well-defined methods, functions, routines, or commands for application software to facilitate the programming languages use of cryptographic or protected resources from an SE that is used as trust anchor

3.3
IoT device
entity of an IoT system that interacts and communicates with the physical world through sensing or actuating

Note 1 to entry: Industrial IoT device is an IoT device which is intended for use by qualified and experience personnel in a controlled manufacturing or processing environment.

Note 2 to entry: For example, industrial uses are smart manufacturing, robot, energy, automobile, transportation, building, and so on. Such an entity can interact with digital twins or asset administration shells of other entities.

Note 3 to entry: An entity can be the combination of the device and a simple sensor. The device comes with a GTA API instance. The sensor does not have an intelligent control unit and does not come with its own GTA API instance.

Note 4 to entry: Industrial IoT devices are not limited to sensor or actuator. Industrial IoT devices include embedded devices such as smart sensor/actuator, programmable logic controller (PLC), edge device (for example, Industrial PC) but not limited to sensor and actuator.

[SOURCE: IEC 60050-741:2020, 741-02-04, modified – The original Note to entry has been replaced with Notes explaining the use of the term in an industrial context.]

3.4
know-how protection
measures supporting the protection against extraction of knowledge or expertise represented by software or other intellectual property from a device

3.5
personality
set of trusted information and cryptographic key material that is used by an application in a specific security context

Note 1 to entry: A personality typically includes the device's own cryptographic material like private keys, secret keys, own public key certificates. A personality can also include information required to establish trust towards partners.

Note 2 to entry: A personality is used within the scope of a specific application. A device can use different personalities in different application contexts.

3.6
public key certificate
set of data that uniquely identifies an entity, contains the public key of the entity, and is digitally signed by a trusted party to bind the public key to the entity

3.7
secure element
SE
component capable of securely hosting functionalities or confidential and cryptographic data or both in accordance with well-defined rules and security requirements

Note 1 to entry: A typical solution for an SE is a one chip microcontroller.

Note 2 to entry: Cryptographic keys are an example of confidential and cryptographic data.

Note 3 to entry: An SE can be realized as pure software component to support future migration to a hardware SE.

Note 4 to entry: An SE can provide special physical protection features such as tamper protection.

3.8
side-channel analysis
exploitation of the fact that the instantaneous side-channels emitted by a cryptographic device depends on the data it processes and on the operation it performs to retrieve secret parameters

[SOURCE: ISO/IEC 17825:2024, 3.9]

3.9
trust anchor
essential security capability that, by definition, must be trusted

Note 1 to entry: A trust anchor can provide provisions to protect the integrity and confidentiality of functions and related information that are required by an application.

Note 2 to entry: The security capability to achieve protection can be provided with an SE. The SE can provide functionality for, for example, secure generation and use of cryptographic key material, and tamper-protected storage of public key certificates.

Note 3 to entry: Data sets, for example, public key certificates starting a certification path, require additional protection against manipulation to be considered as trust anchor. This additional protection can be achieved by, for example, storage in a shielded location.

4 Abbreviated terms

AES	Advanced Encryption Standard
ASN.1	Abstract Syntax Notation One
CA	Certification Authority
CERT	Computer Emergency Response Team
CHAP	Challenge handshake authentication protocol
CO	Component
DCP	Discovery and basic Configuration Protocol
DER	Distinguished Encoding Rules
DH	Diffie-Hellman
DNS	Domain Name System
DSS	Digital Signature Scheme
DTLS	Datagram Transport Layer Security
EC	Elliptic Curve
EK	Endorsement Key
ECDSA	Elliptic Curve Digital Signature Algorithm
FGPA	Field Programmable Gate Array
GDS	Generic Discovery Service
HSM	Hardware Security Module
IACS	Industrial automation and control system
IANA	Internet Assigned Numbers Authority
IDevID	Initial Device Identifier
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPC	Industrial Personal Computer
IPsec	Internet Protocol Security
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6

IIoT	Industrial IoT
IoT	Internet of Things
IT	Information Technology
LDevID	Local Device Identifier
MAC	Media Access Control
M2M	Machine to machine
NIC	Network Interface Card
NV	Nonvolatile
NVRAM	Nonvolatile Random Access Memory
NVindex	Nonvolatile Index
OPC	Open Platform Communications
OPC UA	Open Platforms Communications Unified Architecture
OS	Operating System
PAKE	Password-authenticated key exchange
PC	Personal Computer
PCR	Platform Control Register
PG	Protection Goal
PI	PROFINET International
PKCS	Public Key Cryptography Standard
PKI	Public Key Infrastructure
PKIX	Public Key Infrastructure X.509
PLC	Programmable Logic Controller
PUF	Physical Unclonable Function
RSA	Rivest Shamir Adleman
SC	Security Controller
SE	Secure Element
SEI	Software Engineering Institute
SEP	Secure Element Properties
SHA	Secure Hash Algorithm
SNMP	Simple Network Management Protocol
SoC	System on chip
SSH	Secure Shell
TCG	Trusted Computing Group
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TOFU	Trust on First Use
TPM	Trusted Platform Module
TSS	TPM Software Stack
UCS	Universal Coded Character Set
UTF-8	UCS Transformation Format

5 Architecture

5.1 General

The generic trust anchor application programming interface (GTA API) has been developed to facilitate and implement patterns concerning the security architecture of IIoT devices:

- Trust management process along the lifecycle of IIoT devices (5.5.6)
- Separation of security domains between multiple IIoT applications on a single IIoT device (5.5.1.3, 5.5.7)
- Integration of SEs into the overall security architecture of an IIoT device (5.5.7, 5.6.6)

One of the main objectives of the GTA API is to offload most of the security critical implementation details from the IIoT application developer. The required functionality is encapsulated into a function block that can be maintained independent from the application by security experts.

Annex C provides an in-depth discussion of usage scenarios that are used as a basis for the discussion of GTA API.

Annex E provides some additional examples for the concepts that are introduced in Clause 5.

5.2 Relation to ISO/IEC 30141

The GTA API considers the implementation viewpoint as described by ISO/IEC 30141 [16]. The GTA API provides a high-level application programming interface for the modular integration of SEs into IIoT applications. SEs allow the trustworthy storage and usage of secret and trusted attributes of personalities that are required by IIoT devices to securely interact within an IIoT system. GTA API directly supports the security aspect of the trustworthiness principles for IoT systems that are described in ISO/IEC TS 30149 [17].

5.3 Intended target environment

Architectural decisions for the GTA API are driven by the properties of the intended IIoT and smart manufacturing target environment:

- Embedded devices (such as smart sensors, smart actuators, programmable logic controllers (PLC), edge devices, IIoT devices) with potentially constrained resources
- Unattended use of the device
- Machine to machine (M2M) communication and networked control systems
- "Realtime" operation environments (along with requirements such as deterministic runtime and low latency)

  Realtime is considered to the extent that API design should not inhibit the use of the API under realtime conditions. That means GTA API should support re-entrance and tolerate forced pre-emption by the operating system. It is not assumed that an SE itself meets any special realtime requirements.
- Offline/island mode operation, continuous access to security services such as Public Key Infrastructures is not available, access to those services can be limited to dedicated phases
- Dependable systems in continuous operation mode
- Long-term deployment, systems with long lifecycles

GTA API target environments are characterized by engineered components. Before an IIoT device is put into operation, the device manufacturer, the system integrator, or device operator configures how SEs are to be deployed. The following aspects are typically engineered upfront, for example:

- Secure elements that are in place
- Personalities (including secret or trusted attributes) that are required/used

GTA API especially considers the usage in resource constrained devices in industrial automation and control systems (IACS). Devices in an IIoT environment also include dedicated servers within IACSs.

5.4 Functional scope

Provided functions cover the entire lifecycle of a device, starting from initial device manufacture to its final decommissioning (cf. 5.5.6). Over the lifecycle, functionality to configure, engineer, and manage the security on the device is also needed.

Based on the requirements (cf. Clause C.2) the following functions are addressed:

- Secure Storage for secret and trusted attributes of personalities
- Personality Management and Provisioning
- Entity Authentication
- Key Management for Channel Security
- Data Storage Protection
- SE management

In addition to the nonfunctional requirements listed in Clause C.2 within Table C.13, the following is also addressed:

- Platform independency
- Focus on security design (integration issues like resource arbitration or concurrency are better addressed by platform experts)
- Avoidance of unintentional side-effects ("function f does only work if…")
- Avoidance of functions or properties inhibiting Common Criteria, FIPS-140 certification and the like

5.5 Concepts

5.5.1 Abstraction

5.5.1.1 Architectural abstraction

GTA API provides a modular and scalable architecture.

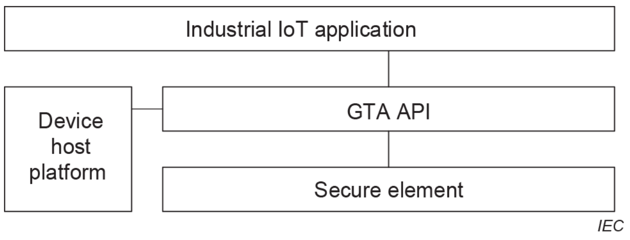


Figure 1 – GTA API environment

The links that are shown in Figure 1 indicate the interfaces exposed by the GTA API framework, which are described within this document:

- Accessing GTA API from the IIoT application
- Embedding GTA API into the host platform
- Adapting towards different types of SE, potentially using technology-specific interfaces, for example, ISO/IEC 7816, Trusted Computing Group TPM Software Stack 2.0 (TCG TSS 2.0)

The focus of this document is on the usability at the application level. The internal API design is mostly left to the implementor of the API. However, this document also provides some guidance on how to build a framework that allows functional extension of the API. Using the API to enable cooperation of different SEs is also addressed.

Figure 2 shows the modular architecture and functional segmentation of the GTA API interface. The individual interfaces shown in Figure 2 constitute the GTA API framework. The function interfaces are detailed in Clause 6.

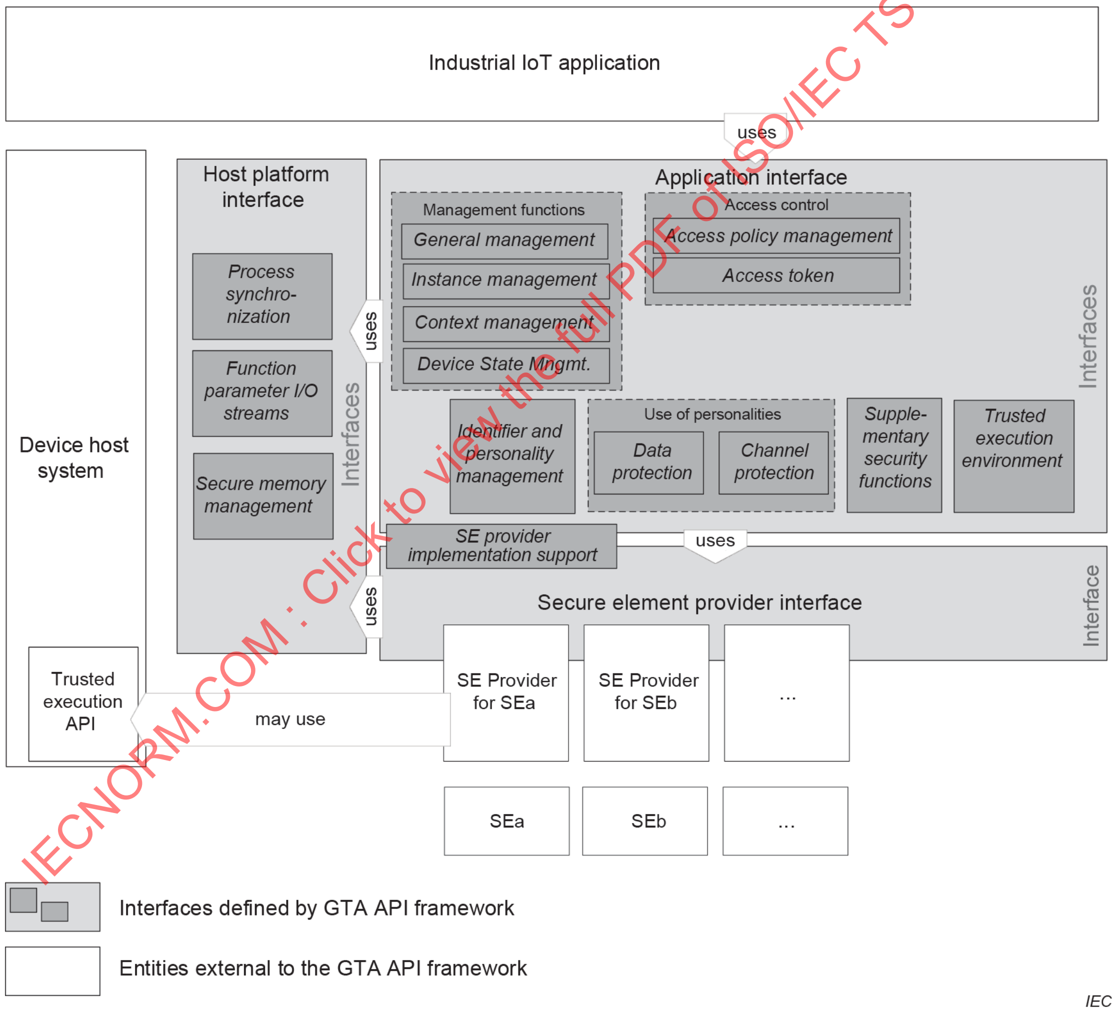


Figure 2 – GTA API modular architecture interfaces

The application interface itself is composed of the following main function groups:

- Management functions to manage the GTA API and used SEs.
- Management of the personalities used to express the identity of a device in a specific trust or security domain. The personality of a device is defined by the secret attributes under the control of the device and additional (trusted) attributes, that is, trustworthy information about other devices. Secret attributes can be, for example, cryptographic secrets, private keys, or passwords. Trusted attributes can comprise, for example, cryptographic verifiers, public keys, or certificates.
- Usage of these personalities to protect data or to establish secure associations towards other IIoT devices.
- Supplementary cryptographic functions which are independent of personalities, for example, random number generation.
- Access control for the resources and functionality offered by GTA API.

The GTA API framework allows transparent integration of multiple SEs into an application. The application developer shall be presented with a unified view of all personalities that are provided by all available SEs. SEs using different technologies and being provided by different vendors are attached by SE providers.

From a conceptual point of view, the API fully abstracts the topology that is used to execute API calls. A functional profile that is used in an API call can require execution within an SE that offers some specific characteristic. Such topological aspects can thus be covered by the profile concept (cf. 5.5.5).

Secure elements are attached using SE providers. These providers implement the functionality required to realize a GTA API function according to a selected profile on a specific SE technology. Depending on the SE technology used, the amount of implementation being either software (that is, as part of the provider) or hardware (for example, encapsulated within a physically distinct SE) can vary (cf. 5.6.1.1). The required security level and existing security constraints depend on the intended execution environment. An application running inside a specific execution environment can desire that specific security properties are provided as native SE functions (cf. 5.5.8). Existing technology-specific APIs can be used to facilitate the implementation of SE providers. Examples are the Trusted Platform Module Library Specification [23] or the ISO/IEC 23465 series on cards and security devices for personal identification [24], [25], [26].

SE providers shall be registered within the GTA API framework before the respective SEs can be used by an application. At the time of registration each provider provides a list of callback functions. These callback functions are used by the framework to dispatch and forwards calls from the applications towards the individual providers. The interface between GTA API framework and provider is defined by the subset of GTA API functions potentially depending on functionality that is provided by the SE. The functional abstraction level (cf. 5.5.1.1) at the GTA API application interface and at the provider interface is the same. While the GTA API framework facilitates interfacing with the host system and contains several functions to maintain auxiliary data structures, the framework does not interfere with functions forwarded to the SE provider. The function set defining the provider interface has been moved to a separate header file (cf. Clause A.3 Provider interface – gta_apif.h).

5.5.1.2 Functional abstraction

GTA API is designed to provide a high-level abstraction interface to cryptographic services. Its functions and interfaces follow a top-down, application driven approach.

Figure 3 shows a typical example for a low-level SE technology-specific crypto API.

```
/*
 * discover public key certifictes and key material
 */

x509_cert = secure_element_read_certificate(
subject_name="/CN=test")

/* ... find matching private key ... */

key_handle = secure_element_get_key(...)

/*
 * setup and perform cryptographic operation
 */

/* ... decide on signature algorithm
 *   based on available key ...
 */

digest = secure_element_hash(
     data = "Hello world!",
     alg = SHA256)

signature = secure_element_sign(
     key_handle,
     digest,
     alg = RSA_PSS)

/* ... do post processing on raw signature ... */
```

Figure 3 – Crypto technology driven API design

This approach provides maximum control over the features and capabilities that are offered by an individual SE. However, usage of such an API requires expert knowledge and creates dependencies towards specific mechanisms and technologies. For example, a signature is not always the best case for data authentication depending on the scenario. Future adaption (for example, use of new algorithms) is often hard to accomplish and costly. This insight is especially true as the lifetime of an industrial application and security mechanisms that are considered to be state-of-the-art is likely to deviate dramatically. It is desirable to be able to exchange substantial parts of the security implementation without an impact to the application.

GTA API aims to hide these implementation details from the application developer. The application developer is able to concentrate on the application. The security experts are able to provide appropriate security according to the intended protection goal (for example, data authentication). Protection goals are addressed by profiles – as discussed in 5.5.5, in a defined trust domain setting – indicated by the personality as discussed in 5.5.4.

Figure 4 shows an example of how functionality like that shown in Figure 3 is expressed using the GTA API.

```
/* open a GTA context to use a personality with a specific
  profile */

h_ctx = gta_context_open(
      personality = "application1",
      profile = "ch.iec.30168.authenticate_basic")


/* compute an authentication tag according to the selected
  profile */
tag = gta_authenticate_data_detached(
      h_ctx,
      data = "Hello world!")
```

Figure 4 – GTA API security service driven API design

Such high-level service calls as exposed by the GTA API are characterized by certain properties like:

- Only the protection goal is specified.
- Detailed information is contained in an application-specific profile.
- Separation is achieved between application logic and cryptographic knowledge.
- Profile configuration can be updated independent of the application.

Application profiles can be specified by experts that combine deep application domain knowledge with security expertise. This expertise is made available to all application developers or integrators through the GTA API.

5.5.1.3 Multi-application capability

Figure 5 shows a scenario where multiple applications use a single SE in parallel. GTA API shall support environments where multiple applications use resources of the trust anchor in parallel. If so, GTA API shall support the separation of access to the SE resources between these applications. GTA API provides functionality to enable the cooperation among GTA API middleware, SE, and the operating system (OS) platform (5.6.5.2, 5.6.6). This claim means access tokens that are acquired by one application are not valid for another application. Also, any content that is processed by one application is not visible to other applications. In case one personality (cf. 5.5.4) is used by different applications, this multiple use shall be properly orchestrated. The separation is transparent for the applications. That means, the different applications are not expected to be aware of the fact that other applications exist and are accessing the SE in parallel.

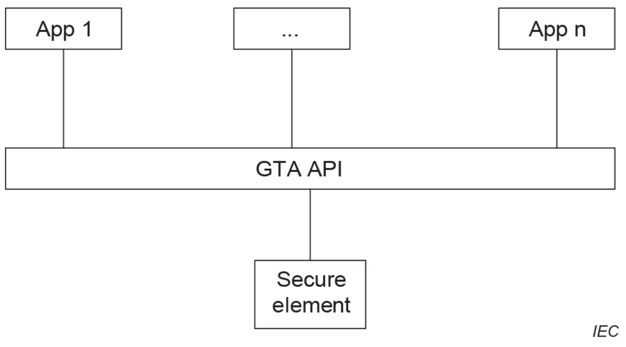


Figure 5 – Multi-application capability

5.5.1.4 Deploying multiple SEs in a device

Sometimes multiple SEs are integrated into a component. The integration of SEs from different vendors into the GTA API framework is supported by dedicated SE providers (cf. Figure 6).

There are different reasons for the use of multiple SEs.

- Cooperation of multiple SE types to accomplish a single security function, for example:
  - An SE enabling secure boot to unlock a separate SE, for example, a software SE implementation using an encrypted software key store
  - An SE providing secure key establishment to make session keys available to a high-bandwidth batch encryption device, for example, an Ethernet network interface card (NIC) with integrated symmetric encryption

    To enable cooperation of multiple SEs, exchange of information between several SEs within the GTA API implementation can be required. This kind of information sharing can be achieved by providing a single SE provider encapsulating the functionality of multiple SEs. Alternatively, the exchange of information between multiple SE providers can be enabled.
- Pooling of multiple SEs of the same type. This SE pool can be beneficial in case a single SE is not able to meet the required performance criteria. From the point of view of functional access, the pool of SEs should act in the same way as a single SE. The internal management of the multiple SEs within the pool is vendor specific and shall be encapsulated inside the SE provider.

Regardless of the topologies of SEs installed in a device, the view that is exposed by the GTA API towards the application programmer shall be uniform. That means, the internal organization is transparent to the application programmer.

This abstraction can be addressed by additional API integration layers, like those sketched in Figure 6.

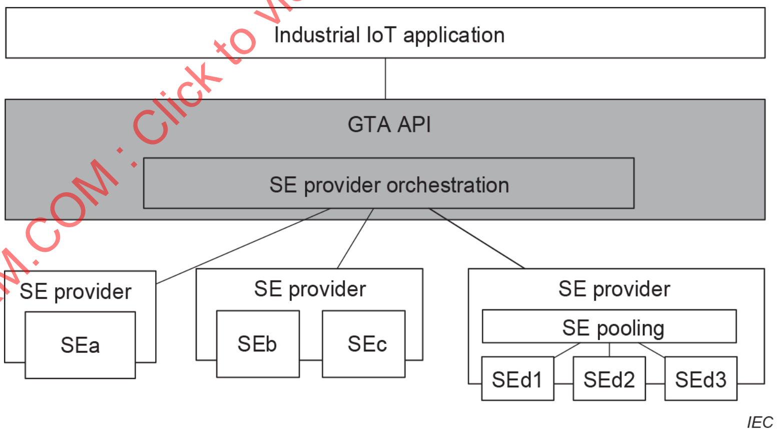


Figure 6 – Secure element abstraction

5.5.2 Object information model

Figure 7 shows the overall object and information model that is used by this document.

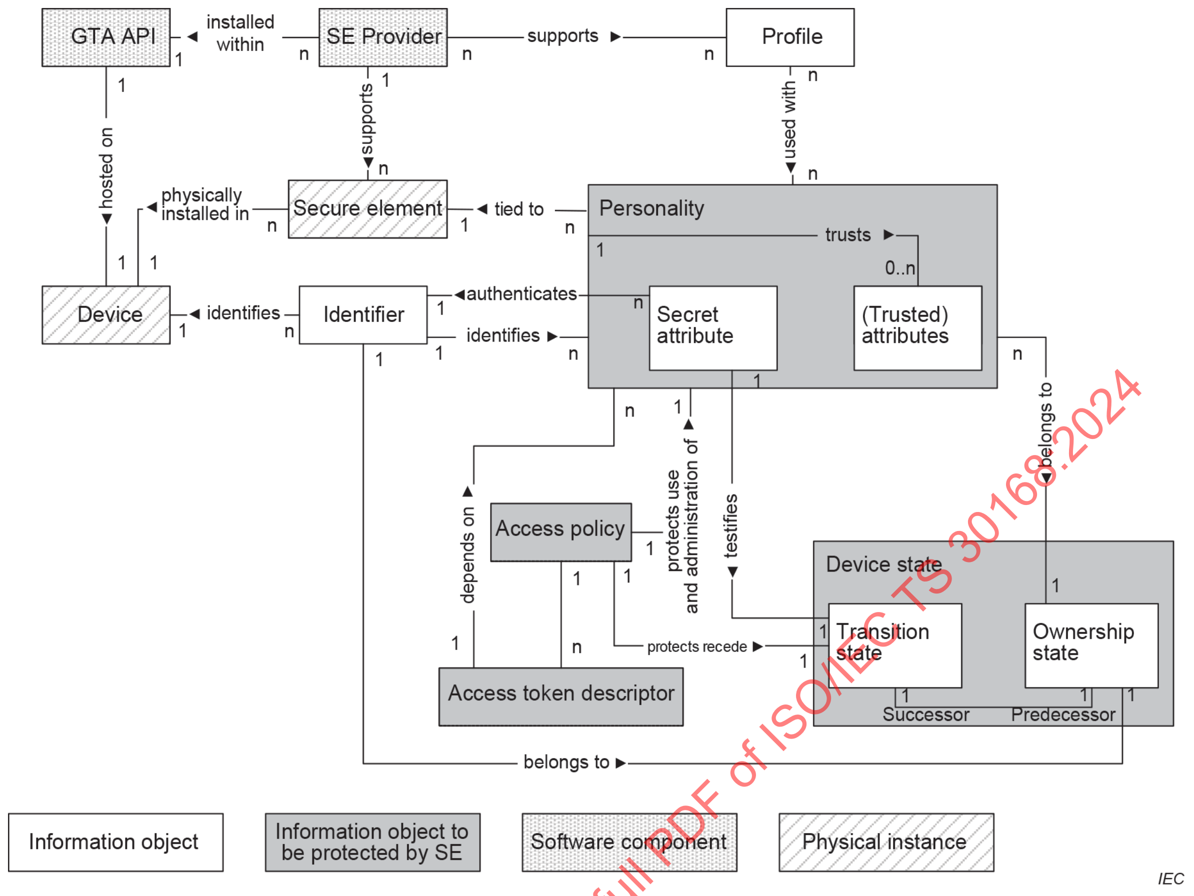


Figure 7 – Object information model (static view)

GTA API is hosted on a device. There is one installation of GTA API for each device.

GTA API may support multiple SEs that are installed in a device. This concept is discussed in detail in 5.5.1.4.

GTA API includes multiple SE providers. Each SE provider allows interaction with a specific type of SE using GTA API. An SE type either refers to a single SE product or a family of SEs.

An SE provider supports different application profiles. The implementor of an SE provider decides whether a specific profile is supported or not. Support for specific profiles also depends on the capabilities of the underlying SE. There can also be distinct SE providers to support multiple profiles for a single type of SE. Further details on profiles are described in 5.5.5.

Various identifiers designated to the device are used to organize and discover personalities. Device identifiers are described in 5.5.3.

Personalities represent the device identity towards the outside world. A personality can act in one of two roles, either as relying party or as requestor. The requestor role uses the secret attributes of the personality to prove the identity of the device. Acting as a relying party a personality relies on trusted attributes to establish trust into a third party. A personality can come with the relying party role only, requestor role only, or act in both roles. Secret attributes are created implicitly during the creation of a new personality and cannot be changed during the lifetime of a personality. Trusted attributes can be dynamically administrated (add/remove/update) during the lifetime of a personality after proper authentication.

Personalities and their attributes are described in 5.5.4 and 5.6.2.1.

Device states are used to control management functions of GTA API, especially for the management of personalities. Each personality belongs to a device ownership state. In case a device state is removed, this operation also removes all personalities belonging to that device state. Device states are described in 5.5.6.

Access policies are used to control the usage of personalities and transition state objects. The access policy describes which conditions are to be met to perform specific operations (cf. Table 1) on the respective object. Access policies are described in 5.6.5.3.

The SE is able to provide the required protection for security relevant information objects. The protection can be achieved by either native mechanisms that are provided by the SE (see also 5.5.8) or by hybrid mechanism. A hybrid mechanism can, for example, use checksums that are maintained on the SE.

Figure 8 shows some additional relations applying to GTA API during runtime. Each application uses its own GTA API instance. The current device state is shared by all instances, that is, in case the device state is changed by an application this change affects all applications. New personalities are associated to the current device state. New personalities can become visible to all GTA API instances immediately after creation or may become visible only after an application opened a new GTA API instance.

Contexts are used to organize operations involving a personality. A context provides an environment for GTA API operations. An operation may extend over multiple individual steps each step being an individual GTA API function call, for example, an enrollment process, establishment of a security association. Contexts can also be used to setup preconditions for operations, for example, setting required attributes or authentication conditions.

Authentication conditions are fulfilled by an application by providing the required access tokens as specified by the respective access policies (see above).

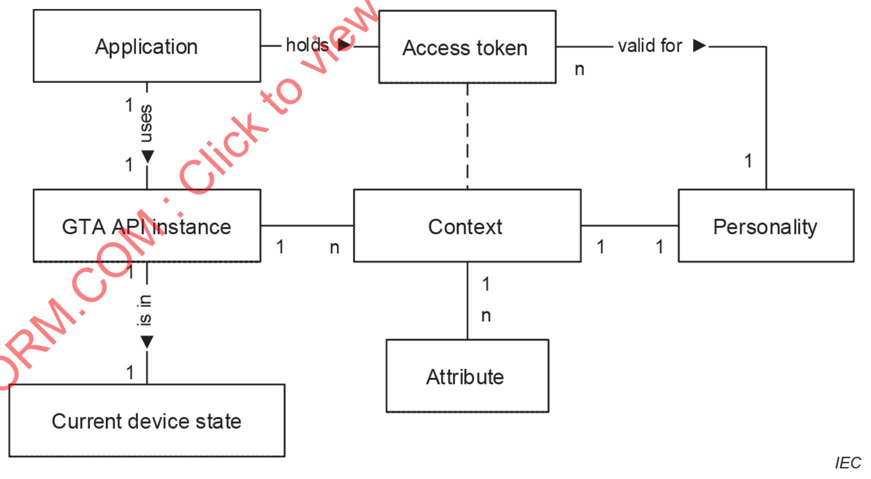


Figure 8 – Object information model (runtime view)

5.5.3 Identifiers

The main purpose of an identifier is to provide a long-term qualifier for the device. This identifier can then be used for comfortable discovery of the device and reference to the device.

There are various identifiers that can be used to identify a device depending on the application, for example:

- A domain name system (DNS) name, Internet Protocol (IPv4 or IPv6) address assigned to the device
- A media access control (MAC) address imprinted into the devices network interface card
- A serial number that is assigned to the device during production
- An immutable hardware serial number that is bound to the device or SE during the manufacturing process

GTA API distinguishes these various identifiers by their identifier type. Identifier types shall be constructed using reverse domain name notation, that is, identifier types start with a registered domain name in reverse order[1], for example, `ch.iec.30168.identifier.dns_name`. This notation allows any entity or organization holding a registered domain name to specify unique identifier types.

The identifier as such does not provide any security assurance. To construct a valid proof of the identity of a device, the identifier needs

a) to be bound to the hardware of the device by some means, and

b) to be mapped to a (usually) cryptographic secret attribute that can be used to compute a verifiable artifact on the identity of the device.

To verify the produced artifact, the relying party needs an authentic assertion about the mapping between identifier and the related secret or trusted attribute. The value of the identifier itself should not depend on the value of related secret or trusted attributes. While these attributes eventually change over time (for example, if a key update occurs), the identifier should persist.

The hardware binding is realized by the properties of the SE. Both the computation and the verification of cryptographic artifacts belonging to an identifier are addressed by the concept of personalities.

An SE or device vendor can provide a set of preassigned, static identifiers which are unalterably linked to the device (for example, a serial number). In case such static identifiers are present, one of them should be of type `ch.iec.30168.identifier.se_generic_hw_immutable`.

5.5.4 Personalities

Personalities represent the device towards the outside world. GTA API shall support an IIoT device to act with different personalities. Personalities can occur in the role of a requestor or relying party.

EXAMPLE Downstream an edge device acts as OPC UA client towards a set of resource restricted devices like sensors and actuators to collect and aggregate data close to the edge device. Edge device and the resource restricted devices use one personality inside this context to form a closed community (security domain). The edge device also communicates upstream as OPC UA server with a datacentre which in turn collects analysed and aggregated data from several edge devices. Here the datacentre is the authority to provide and renew personalities for that closed community (security domain). In general, the aim is to separate security domains by using different sets of cryptographic secret or trusted attributes. Separation of distinct security domains is best practice for risk reduction and separation of responsibilities.

Application programmers can combine personalities with profiles to benefit from these security concepts without having to understand the cryptographic details.

Implementation view specific aspects of personalities are discussed in 5.6.2.

1 See https://en.wikipedia.org/wiki/Reverse_domain_name_notation

5.5.5 Profiles

This document uses functional profiles (that is, mandatory functionality of the GTA API instance and its environment) to enable specific scenarios occurring in an IIoT environment.

Instead of specifying cryptography and SE specific individual parameters, the application programmer can specify the respective scenario profile that is required for a specific function. The GTA API ensures that these profiles are realized in an interoperable manner between different implementations. Detailed information – such as algorithms used, key sizes, encapsulation formats – is encapsulated and hidden by the profiles.

Examples for functional scenario profiles are:

- Binary image and data measurement, verification, and authentication to support device and application integrity
- OPC UA authenticated communication
- PROFINET authenticated communication
- Secure communication for substation automation systems

A profile identifies a distinct usage scenario, for example, OPC UA. Profiles encapsulate the complexity of the cryptographic base operations and hide them from the application developer. For example, to compute an authentication signature, instead of selecting a specific cryptographic signature algorithm, digesting scheme, padding, and encapsulation scheme, the application developer straightforwardly selects a profile. The profile was agreed upon by domain and security experts to be suitable for the intended tasks.

Profiles allow the application programmer to benefit from the expertise of security experts. Application programmers are not expected to understand and deal with the details of different types of cryptographic mechanisms. If both peers use the same profile, conformance to the profile ensures the interoperability among peers.

Profiles are identified by profile names as described in 5.6.3.1. Definitions of a profile should contain a clear definition of the intended use case to help the application programmer to select the correct profile. The scenarios described in Clause C.1 can be used as a reference.

Several basic profiles are defined as part of this document within Annex B. These basic profiles are intended to support functions within the GTA API framework (for example, access control). The basic profiles also facilitate some basic use cases (for example, device local data protection).

The basic profiles that are given in Annex B can also be used as examples for the definition of additional profiles. The following aspects should be considered when defining profiles:

- Functional abstraction level: It is possible to define profiles on different abstraction levels. Profiles can be defined on a high level as, for example, the profile ch.iec.30168.basic.local_data_integrity_only given in Clause B.2. The profile ch.iec.30168.basic.local_data_integrity_only leaves a lot of freedom for the provider implementation if the semantic intention is maintained. In contrast, there can be narrowly defined profiles like ch.iec.30168.basic.passcode as given in Clause B.1.
- Interoperability: The profile should take into account the level of interoperability that is required between different implementations for the same profile. This is important in case cryptographic artifacts (for example, protected data) are to be exchanged between devices or other external parties using different implementations. In contrast to this, ch.iec.30168.basic.local_data_integrity_only (see Clause B.2) is an example for a profile which does not require any interoperability among implementations.

- Minimum functionality: GTA API should provide the minimum functionality that is required regarding the encapsulation of SE-specific functionality. GTA API profiles should not be designed with extensive handling of container formats and cryptographic protocols. This handling is expected to be done by the application or a format or protocol-specific SDK using GTA API to resort to dedicated trust anchor functionality.
- Closure: A profile should include a well-defined functionality. The delivered artifacts are ideally already defined by other standards. Wherever possible, it should be avoided that GTA API and other code are required to manipulate the same artifact, although this is often not completely achievable for legacy protocols.

Implementation view specific aspects of profiles are discussed in 5.6.3.

Clause G.2 provides an example for the usage of a basic profile.

5.5.6 Device states

Device states are used to manage objects such as identifiers and personalities that are created by different device owners. Typically, the device state changes as the device moves from one owner to another along the value creation chain. Examples are the transition from the initial device manufacturer to a machine builder integrating the device into a more complex machine, and finally to the operator deploying the machine into the shop floor. Figure 9 illustrates this value creation chain. Stakeholders and related device states that are often considered within IIoT are used as an example.

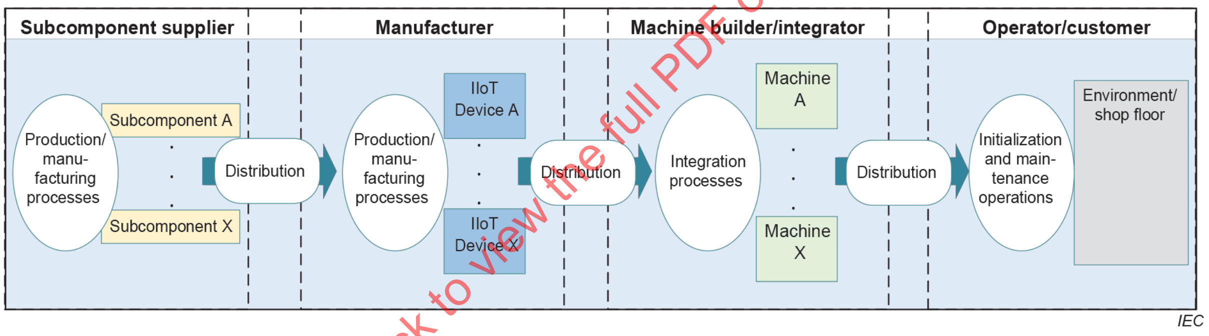


Figure 9 – Value creation chain

Each owner wants the possibility to create its own set of objects on the SE. However, it is important to ensure that subsequent owners are not able to tamper with the settings already installed. It shall also be possible to reset a device to a well-defined state, for example, in case it is moved to a new operational environment.

Implementation view specific aspects of device states are discussed in 5.6.4.

5.5.7 Access control

The execution of several operations on specific objects that are performed through GTA API on an SE are subject to access control. Table 1 lists the objects and the applicable usage restrictions.

Table 1 – Access control

Object	Usage	Description
Personality (5.5.4)	Use	Usage of the personality involving a cryptographic computation with a secret attribute
	Admin	Modification of a personality (activation/deactivation) or a trusted attribute (addition/removal), for example, a CA public key certificate, of the personality
Transition device state (5.5.6)	Recede	Recede from a device state to a previous device state; Usage control is determined by the device state to be removed.

Usage restrictions for an object are defined with access policies at the time of object creation and cannot be changed afterwards.

Before using an object that is protected by an access policy, the access policy shall be satisfied by either

- meeting an implicit access condition, or
- providing an access token.

Access conditions depend on the state of GTA API and its runtime environment. Whether an access condition is met or not cannot be influenced by the application attempting to use an object.

Access tokens must be actively acquired and provided by an application attempting to use an object.

Implementation view specific aspects of access control are discussed in 5.6.5.

5.5.8 Secure element properties

This subclause only describes possible properties for SEs that can be used in the context of the GTA API. It does not mandate any of these properties. The intention is to describe specific SE properties (SEPs). These SEPs can be used within other clauses or subclauses of this document or within other documents referencing the GTA API. As an example, they could be used to specify security levels for GTA API specifics like personalities. SEPs can also be used in the context of security classes and attestation (see Annex D). From a GTA API point of view, it is intended to use them as parameters when creating personalities.

Whether and to what extent these properties are met by a specific implementation of GTA API in combination with a specific SE is out of scope of this document. The properties achieved by a specific implementation can be addressed by a security evaluation of the resulting system. Common evaluation processes are IEC 62443 (https://www.iec.ch/blog/understanding-iec-62443) and Common Criteria (https://www.commoncriteriaportal.org/).

At first, a foundation for defining SE properties is given by relevant protection goals that are potentially supported by SEs in some way. The following basic protection goals (PGs) are seen as relevant in the context of the GTA API and the underlying SEs:

- PG1a: Protect integrity of secrets, trusted attributes, and security metadata stored by SEs, against offline attacks or when powered off.
- PG1b: Protect confidentiality of secrets, which are stored by SEs, against offline attacks or when powered off.
- PG1c: Protect access to critical SE data by simple means, that is, based on identifiers, passwords, or platform configuration register (PCR) values.

- PG1d: Protect access to critical SE data by sophisticated means, that is, based on corresponding GTA API access control mechanisms.
- PG2a: Protect against exposure of private keys, which are stored by SEs, during online or runtime attacks.
- PG2b: Protect against replay of critical SE communication messages, for example, by considering and enforcing cryptographic freshness.
- PG2c: Protect validation of trusted attributes, for example, compare password or verify public key operations, and corresponding access right decisions, which are done by an SE, against manipulation.
- PG2d: Protect operations affecting security metadata, especially transitions between device states.

NOTE Security metadata includes, for example, rules for transition between device states (cf. Figure 11)

Although implicitly grouped by their identifiers, protection goals can be treated individually. Nevertheless, a combination of protection goals is always possible.

How an SE achieves one of the protection goals can be confirmed by attestation that also uses the GTA API (cf. D.2).

From a technological point of view, the goals that are preceded by PG1 mainly correspond to an appropriately secure memory. The goals that are preceded by PG2 mainly correspond to a trusted execution environment. PG1c, PG1d, and PG2b need additional technological measures.

The following SE properties that are defined for the context of the GTA API are used to address the protection goals given above:

- SEPintegri: Protection of integrity of secrets, trusted attributes, and security metadata against offline attacks
- SEPintpers: Protection of integrity of a personality set, that is, the following mappings:
  - Between secret and trusted attributes
  - Between secret attributes, trusted attributes, and personalities
  - Between personalities and device states
- SEPintmeta: Protection of security metadata
- SEPseccrea: Securely create secret attributes on an SE
- SEPsecread: Ensure that secret attributes cannot be read from an SE when powered off
- SEPauthuse: Protection of use of secret and trusted attributes on an SE by using access tokens
- SEPauthman: Protection of management, for example, addition, of secret and trusted attributes by using device states
- SEPauthtru: Protection of validation based on trusted attributes
- SEPsecextra: Ensure that secrets cannot be extracted from an SE during operations that use them
- SEPsecrepl: Ensure that security-critical communication messages cannot be replayed

Table 2 gives a mapping between SE properties and protection goals.

Table 2 – Mapping between SE properties and protection goals

SEP	PG1a	PG1b	PG1c	PG1d	PG2a	PG2b	PG2c	PG2d
SEPintegri	X							
SEPintpers	X							
SEPintmeta								X
SEPseccrea		X						
SEPsecread		X						
SEPauthuse			X	X				
SEPauthman			X	X				
SEPauthtru							X	
SEPsecextra					X			
SEPsecrepl						X		

An SE property or a set of SE properties is characterized by Table 2. An SE can state to, for example, just achieve PG1c by SEPauthuse. An SE can announce to have further security properties regarding the protection goals mentioned or can even meet further protection goals. It can also be stated by an SE through attestation whether its security properties have been assessed or tested according to certain standards.

5.6 Implementation view

5.6.1 System design considerations

5.6.1.1 Secure element hardware-software co-design

An implementor of the GTA API can decide which functionality is provided by the actual SE hardware and which functionality is provided by the SE provider software.

Depending on the security requirements of the application, different types of SE and respective SE providers are possible:

- Software only, for example, to enable later migration to hardware based SEs.
- Hardware integrated, for example, firmware trusted platform module (TPM), trusted execution environment (TEE), system-on-chip (SoC) integrated, dedicated field programmable gate array (FPGA), security controller (SC), physical unclonable function (PUF), or hardware security module (HSM).
- Functional building blocks (for example, network element) that offer integrated security functionality (for example, authenticated network bulk encryption).

  This document does not mandate any specific technology for interfacing between the SE provider and the SE (cf. Figure 2). Vendors should make sure that the communication between SE provider and SE is sufficiently protected regarding the intended operational environment. Vendors should specify their assumptions for the intended operational environment. If specific technologies (for example, ISO/IEC 7816 Secure Messaging) are used which require keeping a secret at both ends of the communication channel, these technology details shall be handled completely within the SE provider. It is assumed that the IIoT application and GTA API (respectively the SE provider) are hosted within a common runtime address/process space. Following this assumption no extra security is achieved by coercing the application to care about the security of communication that is related to the SE.

5.6.1.2 System security considerations

This document describes an interface. Its design was chosen to support secure use of an implementation of the GTA API. For the security of a system using such an implementation, a holistic approach is recommended. The security architecture of systems using a GTA API implementation should be created following a secure product development lifecycle, the result from a security risk assessment, and a list of relevant standard security requirements for a desired security level. For example, the International Standards [19], [20], [21], and [22] provide further guidance.

5.6.2 Personalities

5.6.2.1 Personality attributes

A personality is used within one or more specific security domains and enables the use of different security services to achieve, for example:

- Authentication of a GTA API instance as a whole or in part
- Verification of the authenticity of a GTA API external instance

Thus, a personality can include a specific secret that is used to prove a specific claimed identity. A personality can also include specific authentic information to verify if some secret is known by an external entity.

For the GTA API, a personality is defined using the following kinds of attributes:

a) Secret attributes
b) Trusted attributes
c) General attributes

The following examples are given to illustrate the personality concept.

The first example is a personality to be used within a public key infrastructure X.509 (PKIX) trust model which contains the following attributes:

- Secret attribute: An RSA private key used to compute signatures to prove the authenticity of the device.
- Trusted attributes: Peer public key certificates being unconditionally trusted by the device. These certificates include self-signed public key certificates, certification authority (CA) root certificates, and pinned peer public key certificates.
- General attributes: RSA public key certificate corresponding to the devices private key.

The second example shows a personality using an asymmetric crypto protocol without a public key infrastructure (PKI), for example, secure shell (SSH):

- Secret attributes: Set of private host keys, typically an RSA, a DSS, and an ECDSA key
- Trusted attributes: List of public keys for known hosts

The personality that is given in the last example is intended for a hybrid approach like a password authenticated Diffie-Hellman (DH) key exchange:

- Secret attribute: Password used by the device to authenticate itself.
- Trusted Attributes: Password verifier used to authenticate a password provided by a peer in a password-based authentication protocol.
- General attributes: DH group parameters to be used in a password authenticated key exchange.

Trust lists as defined by the OPC Foundation [15] are a combination of trusted attributes. Trust lists include information, such as trusted public key certificates or trusted certificate revocation lists, and general attributes.

Both secret and trusted attributes shall be protected according to the specified SE protection properties (cf. 5.5.8). An SE can provide an execution environment able to run operations using secret or trusted attributes as input. Confidentiality protection of trusted attributes is most probably not needed in contrast to confidentiality protection of secret attributes. Both types of attributes usually need integrity protection.

5.6.2.2 Personality fingerprint

Each personality shall define a well-defined process to compute a fingerprint on the personality. The fingerprint shall provide a cryptographically strong bond to the personality (for example, to refer to the personality in a cryptographic protocol to ensure cryptographic binding). In case the attributes relating to the personality change, the fingerprint shall also change.

Especially in case a personality is deleted and re-created using identical application provided identification attributes (for example, personality name), the fingerprint of the new personality shall be distinct from the deleted one. Otherwise, it would be possible to compromise the security by replacing a personality with a new personality which provides weaker security properties.

An example for a personality fingerprint is a cryptographic checksum (for example, based on SHA256) computed over a well-defined part of the value of the personality.

5.6.3 Profiles

5.6.3.1 Profile naming

Profiles shall be uniquely identified. GTA API uses string type profile names for this purpose. To avoid naming conflicts, the existing IANA Internet domain name registration system (DNS) shall be used for namespace management. Profile names shall use reverse domain name notation, that is, profile names start with a registered domain name in reverse order[2]. This notation allows any entity or organization holding a registered domain name to specify a profile, for example:

- ch.iec.30168 for profiles defined within this document
- org.opcfoundation for profiles defined within the scope of OPC UA
- org.ietf.tls13 for profiles defined within the scope of transport layer security (TLS) 1.3

The naming scheme can be used to define subprofiles, for example:

- org.ietf.tls13.x509 for X.509 public key certificate-based TLS
- org.ietf.tls13.psk for PSK (pre-shared keys) based TLS

5.6.3.2 Personality creation and deployment profiles

Personality creation and deployment profiles are used during the creation and deployment of new personalities.

The properties that shall be determined by a personality creation and deployment profile are defined in Table 3 and Table 4. Profile definitions shall provide any mandatory (m) properties. Optional (o) attributes can be specified to provide additional information.

2 See https://en.wikipedia.org/wiki/Reverse_domain_name_notation

Table 3 – Properties of personality creation profiles

Property	Mandatory (m)/ Optional (o)	Description
Security Mechanism	m	Type and strength of the security mechanism that is supported by the personality (for example, AES256 secret key, EC25519 private key)
Fingerprinting	m	Scheme or algorithm used to derive a unique personality fingerprint (5.6.2.2)
Attributes	o	Well-defined attributes of the personality
Usage Info	o	Reference to enrollment (5.6.3.3) or usage profiles (5.6.3.4) that is supported by the resulting personality

Table 4 – Properties of personality deployment profiles

Property	Mandatory (m)/ Optional (o)	Description
Security Mechanism	m	Type and strength of the security mechanism that is supported by the personality (for example, AES256 secret key, EC25519 private key)
Import Format	m	Type and format of the archive data that are used to represent the personality being imported (for example, PKCS#12)
Fingerprinting	m	Scheme or algorithm that is used to derive a unique personality fingerprint (5.6.2.2)
Attributes	o	Well-defined attributes of the personality
Usage Info	o	Reference to usage profiles (5.6.3.4) that are supported by the resulting personality

5.6.3.3 Personality enrollment profiles

Personality enrollment profiles are used during the enrollment and registration of personalities using a trusted third party (for example, an X.509 certification authority).

The properties that shall be determined by a personality enrollment profile are defined in Table 5. Profile definitions shall provide any mandatory properties. Optional attributes can be specified to provide additional information.

Table 5 – Properties of personality enrollment profiles

Property	Mandatory (m) Optional (o)	Description
Profile Dependencies	m	The profile may require that the personality has specific properties that are typically specified during personality creation. These dependencies can be expressed by referencing to a specific personality creation profile. It is also possible to determine an appropriate creation profile from the specification of the supported security mechanisms.
Enrollment Attributes	m	Attributes (including attribute format) that shall be provided to build the enrollment request (for example, an X.509 subject name, requested X.509 public key certificate extensions)
Enrollment Artifact	m	Format of the enrollment artifact (for example, PKCS#10)

GTA API profiles enable service providers for X.509 certification/registration authorities to offer services that are aligned with enrollment profiles defined in association with GTA API. In this case the profile can also consider properties which are not directly related to GTA API. One example property is the template used for the resulting X.509 certificate such as subject name attributes, validity, or certificate extensions. This offering would allow users to decide whether a specific CA service is suitable for an intended application based on the profile name as single criterion.

5.6.3.4 Personality usage profiles

Personality usage profiles are used while applying personalities for security operations. Examples for security operations are creating a signature, performing encryption or decryption, computation of cryptographic checksums, verification of secret passwords.

The properties that shall be determined by a personality usage profile are defined in Table 6. Profile definitions shall provide any mandatory properties. Optional attributes can be specified to provide additional information.

Table 6 – Properties of personality usage profiles

Property	Mandatory (m) Optional (o)	Description
Profile Dependencies	m	The profile can require that the personality has specific properties, typically specified during personality creation or deployment. These dependencies can be expressed by referencing to a specific personality creation or deployment profile. It is also possible to determine an appropriate creation or deployment profile from the specification of the required security mechanisms.
Supported Functions	m	Functions of GTA API supported for this profile. It is possible that some profiles will not be applicable for use with all functions of GTA API. For example, it is possible that a profile that is destined to support the establishment of secure channels between two parties will not work with data protection functions.
Usage Attributes	m	Extra attributes required for usage of the personality, for example, a challenge value required within a challenge-response protocol
Usage Artifact	m	Type and format of the artifact resulting from an operation involving the personality (for example, PKCS#1); Depending on the profile, multiple artifacts can result while using a personality according to the given profile.

5.6.4 Device states

5.6.4.1 General

GTA API distinguishes two types of device states:

- Owner states: The device is in the ownership of one actor along the value creation chain (for example, the machine builder).
- Transition states: The device is in transition from one owner to the succeeding owner. The transition states are created by the originating owner to allow transition to the next owner in the chain. GTA API allows testifying the genuineness of a device transition state. Successful attestation of a transition state allows the new owner to decide whether the device is still in the state that was intended by the originating owner. A failed attestation indicates that someone else already tampered with the device. Once the transition state gets verified, the recipient takes ownership of the device by installing the owner state of the recipient on top of the latest transition state.

The organization of device states can be looked at as a stack. Two operations are possible on the device state stack (cf. Figure 10):

- Advance: Pushes a new state on top of the existing stack. The new state on top of the stack is either a transition state or an owner state.
- Recede: Pop states from the top of the stack until the next transition state is reached. All personalities belonging to ownership states that are affected by the operation are discarded.

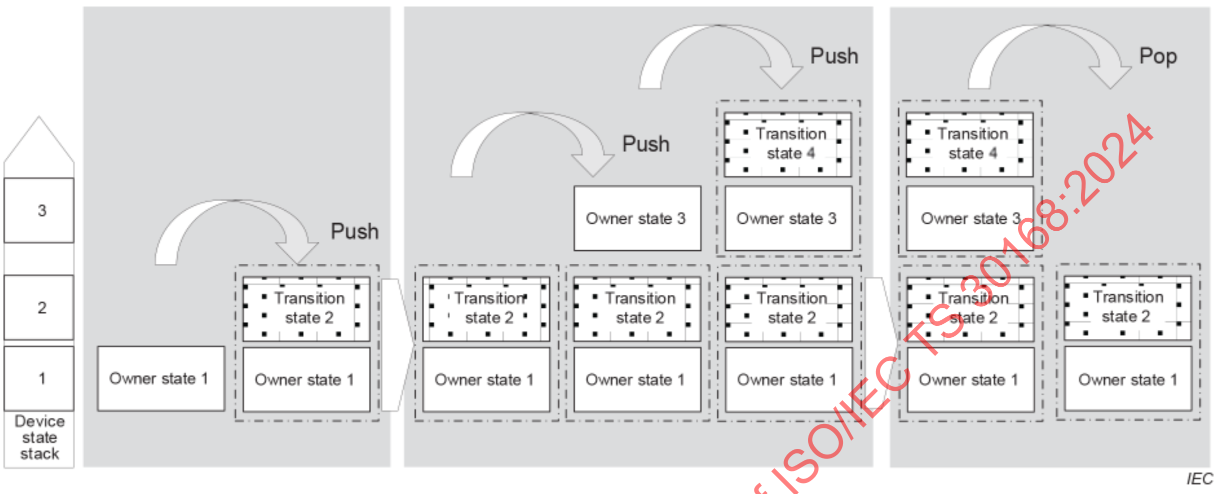


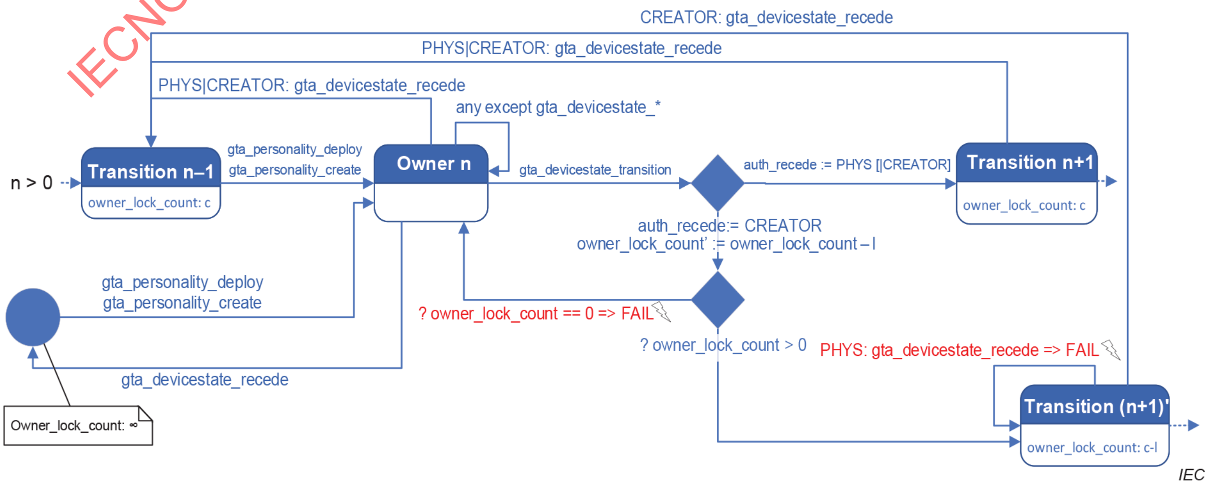
Figure 10 – Device state stack

Push and pop operations result in transitions between device states as shown in Figure 11.

A new device state is pushed automatically whenever a new personality is created from either the initial state (cf. 5.6.4.2) or a transition state.

Transition states are created by an explicit invocation of a `gta_devicestate_transition()` operation. This operation also allows definition of an access policy which shall be met to remove the transition state from the stack:

1) `PHYS`: The removal requires physical presence (cf. 5.6.5.2.5).
2) `PHYS|CREATOR`: The removal requires either physical presence or explicit authentication as defined by the state creator.
3) `CREATOR`: The removal requires explicit authentication as defined by the state creator, that is, the state cannot be removed using physical presence.



CREATOR: gta_devicestate_recede
PHYS|CREATOR: gta_devicestate_recede
PHYS|CREATOR: gta_devicestate_recede
any except gta_devicestate_*
n > 0
Transition n–1
owner_lock_count: c
gta_personality_deploy
gta_personality_create
Owner n
gta_devicestate_transition
auth_recede := PHYS [|CREATOR]
Transition n+1
owner_lock_count: c
auth_recede:= CREATOR
owner_lock_count' := owner_lock_count – l
gta_personality_deploy
gta_personality_create
? owner_lock_count == 0 => FAIL
? owner_lock_count > 0
PHYS: gta_devicestate_recede => FAIL
gta_devicestate_recede
Owner_lock_count: ∞
Transition (n+1)'
owner_lock_count: c-l
IEC

Figure 11 – Device state transitions

The use of option 3) can be restricted by an additional parameter `owner_lock_count`. `owner_lock_count` determines whether creator-defined-only authentication can be specified for future states. Whenever a new transition state is pushed specifying option 3), the value that is specified for `owner_lock_count` shall be smaller than for preceding push operations. In case no smaller value exists (that is, `owner_lock_count == 0`) option 3) can no longer be used, that is, all subsequent states can be removed using physical presence.

The following example illustrates how GTA API supports the transition of ownership between different entities.

Figure 12 shows the transition from the device manufacturer to the machine builder. Initially the device is in state 1. State 1 is used to provision the personalities of the manufacturer to the device. After that the device manufacturer pushes a transition state. The machine builder verifies the genuineness of the transition state and continues installing device state 3. Device state 3 is owned by the machine builder and used to install the manufacturer personalities.

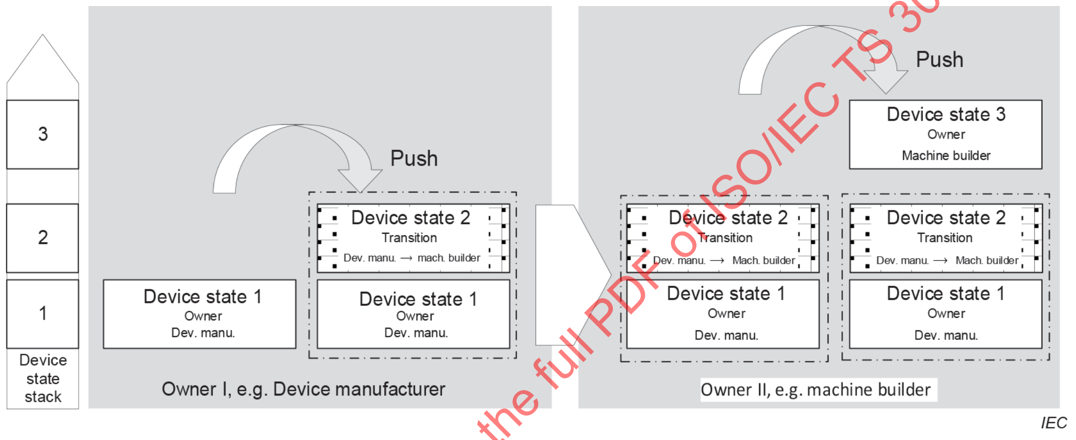


Figure 12 – Device state stack (push)

Figure 13 illustrates how the device is returned to the device state owned by the machine builder after a machine has been returned from operation (decommissioning). Typically, the machine builder does not know any secret attributes that have been installed by the operator. However, it is possible that the machine builder is still allowed to use physical presence to pop the device state that has been installed by the operator. This operation returns the device to the device state created by the machine builder.

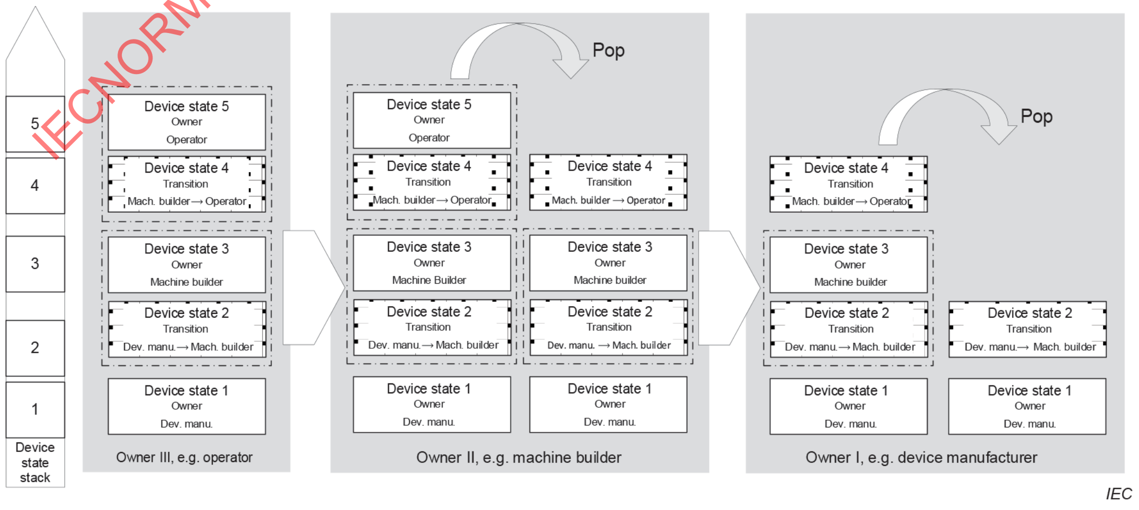


Figure 13 – Device state stack (pop)

After the last pop operation permitted by GTA API, one residual state which cannot be popped remains on the stack. Whether pushing the first device state requires some means of authentication is left to the implementation (cf. 5.6.4.2).

5.6.4.2 (Pre-)Initial device state

The initial device state of the GTA API from a security management point of view strongly depends on its specific integration and initial configuration of SEs. It includes:

- Primary access conditions
- Primary personalities (for example, including trusted attributes for further bootstrapping)

Primary access conditions and in primary personalities may be empty after manufacturing or even at later stages. Apart from the device state, that is, access conditions and provided personalities, the functionality of the GTA API is also determined by the number of different profiles offered. Security levels are understood as part of the initial device state of the GTA API. But because security levels are static attributes of the functionality that is offered by the GTA API, they are not explicitly covered in 5.6.4.2.

The overall trustworthiness of the initial state of the GTA API depends on the following presuppositions:

- Pre-initial state of the component or device
- Pre-initial states of all SEs at the time of the first installation of the GTA API
- Physical and logical protection of the component or device that serves as installation target for the GTA API
- Physical and logical protection of the physical environment in which the component or device is used at the time of the first installation of GTA API
- Protection of the GTA API installation files before they are used and at their time of usage
- Protection of any further item that is later used for contributing to the initial state of the GTA API, for example, states resulting from secure boot

The initial device state of the GTA API can be created based on different trust models. Here, the following two fundamentally different trust models are considered:

a) Trust on first use (TOFU), also referred to as "resurrecting duckling"

   NOTE "Resurrecting duckling" is a security policy mainly based upon the two states "imprintable" and "imprinted". An imprintable device accepts any security configuration from any new owner. An imprinted device acts according to the security configuration from its owner. F. Stajano and R. Anderson [29] use the definition: "the resurrecting duckling security policy model [...] describes secure transient association of a device with multiple serialised owners". The latter essentially means that a device can recede to the state imprintable.

b) Trust that is based on already possible protection based on personalities

Trust that is based on already possible protection that is based on personalities is only possible if at some earlier point in time TOFU was performed. A common place for TOFU for a GTA API instance is during manufacturing of the component that uses the GTA API. As preparatory step, manufacturers of SEs can perform TOFU on their own. This step provides the component or device manufacturer with a way to authenticate against the SEs on first usage. But TOFU can also be performed by other players like distributors, machine builders or operators. After TOFU has been performed once, all following steps can be designed in a way that allows to build up a full chain of trust. This design relies on a consistent use of personalities and access policies that have been provided by the first TOFU step. A high level of trustworthiness can be reached when TOFU is performed during manufacture of SE and GTA API component or device and when all subsequent steps are consistently implemented in secure and reliable ways on top of this first step.

As an example, trust that is based on manufacturer protection and personalities including manufacturing of SE is now described further and briefly elaborated on. Other trust models can be used as well as other ways to extend the trust provided by manufacturer personalities.

Assume some component that offers applications that in turn use the GTA API together with exactly one SE in form of a security controller. Further assume that the security controller is manufactured by one manufacturer and the rest of the component is manufactured by another manufacturer. The manufacturer of the security controller is referred to as ‘manufacturer SC’, that is, manufacturer security controller. The manufacturer of the rest of the component is referred to as ‘manufacturer CO’, that is manufacturer component. Manufacturing includes physical and logical parts, that is, hardware and software or firmware. In this setting, manufacturer SC uses a protected manufacturing environment to store some secret encoded as reference value within the secure memory of the security controller. The security controller gets locked by manufacturer SC and can only be opened using this secret. Using some previously established trusted communication channel or trusted delivery path, manufacturer SC delivers the secret or the corresponding security controller to manufacturer CO. Manufacturer CO integrates this security controller within a protected manufacturing environment. Here, integration means hardware assembly of the component. In this assembly process the security controller is one element that gets integrated together with first installation of the GTA API from a protected software repository. During GTA API installation, the secret for unlocking the security controller is provided as well. The GTA API is used by a corresponding application to first unlock the security controller. The application then generates an asymmetric cryptographic key pair on the SE together with a certificate signing request for the public key part. As last step the application imports a manufacturer X.509v3 public key certificate together with corresponding trusted attributes. All artifacts created by the application altogether serve as initial device identifier (for example, initial device identifier (IDevID) according to IEEE 802.1AR [10]). During this process, the GTA API pushes Device State 1 on its stack representing the IDevID with manufacturer CO as owner. The IDevID is the first personality that can be used by the GTA API, for example, to update the GTA API itself under control of an update application. Such updates can include updates for SE and SE provider whereby security can in turn be based on the pre-initial states of the SE. Next, component management pushes Device State 2 as transition state on the GTA API device state stack. Again, a secret attribute can be created and stored within the security controller in a way that unlocking of the GTA API is only allowed after proving knowledge of this secret by an application, which can again include setting of trusted attributes. As an alternative, an application is provided in a way that Device State 2 is only allowed to be used once by the next owner. That means, multiple use of the transition state is not allowed and can be detected. Setting of Device State 2 concludes the GTA API specific runtime part of manufacturing. All further transitions can as well be protected using adequately created transition states or device states. Delivery of unlocking secrets, if used, from one owner to the next should be protected using appropriately trusted out-of-band ways. This way, new device states can only be pushed onto the device state stack by respective component users. If a transition state can only be used once, a new owner can detect whether the GTA API is still in this state. If transition states are created without any protection, the GTA API can be configured further by anyone who has adequate access to the component that hosts the GTA API.

As a complement, initial access that always starts empty either stays empty or is based on a specific set of previous operations. Conducting secure boot is an example for such operations. The latter means that depending on reference values for unlocking secrets, software or firmware image hashes, initial access can be achieved during start-up of the component or device.

After all steps mentioned before were performed, the component or device with integrated GTA API has an initial and potential one or more further device states of the GTA API. These device states include personalities and sets of secret and trusted attributes. There are defined conditions for initial access of GTA API access control depending on functionalities of the component during start-up. Summed up, beginning with manufacturers SC and CO all subsequent owners can use the GTA API in a trustworthy way that is determined by the security policy of the preceding owner.

Refer to 5.6.4.1 and 5.6.5 for further descriptions on how device states and initial access, respectively, are reached or managed. Also refer to Clause E.1 for a specific example.

5.6.5 Access control

5.6.5.1 Initial access

Initial access shall be available automatically after successful start-up of the system (including GTA API and SE). The conditions which apply to grant initial access are implementation dependent. These conditions shall be documented for the GTA API user developing an application on the device. Examples for such conditions are:

- Completion of a secure boot sequence
- Completion of specific system runtime integrity check or system environment integrity check, or both.

Initial access can be void if there are error conditions that are detected by either GTA API or the SE. The respective error conditions are implementation-specific and shall be documented.

5.6.5.2 Access tokens

5.6.5.2.1 General

All access policies that extend beyond initial access are granted using access tokens. Figure 14 illustrates how an access token is built to incorporate several properties.

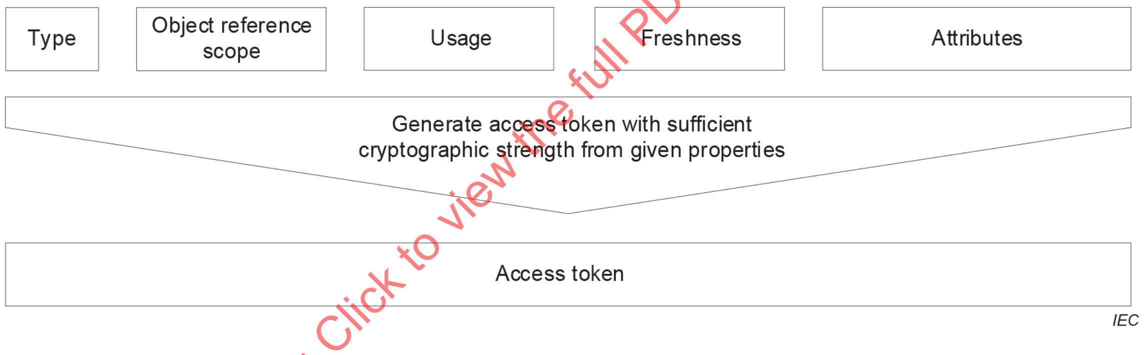

Type
Object reference scope
Usage
Freshness
Attributes
Generate access token with sufficient cryptographic strength from given properties
Access token
IEC

Figure 14 – Access token

An access token confirms the following properties.

- Type: There can be different types of access tokens. Access tokens allow to convey privileges among local processes or allow to grant privileges that are held by users or remote services. Access token types are described in detail in 5.6.5.2.
- Object Reference/Scope: A reference to the object or a set of objects for which the token is valid (cf. Table 1).
- Usage: The kind of permission granted (cf. Table 1).
- Freshness: Information allowing to determine whether the token is still valid. The validity of an access token shall not persist after restart of the device.
- Attributes: Extra attributes which allow the SE to determine whether the token is valid for the intended operation. Each different type of access token contains specific attributes. Attributes can be used to tie the applicability of a token to precisely defined operations. An example for such an operation is the update of a trusted attribute to a specific value. Tying tokens to operations is especially useful for personality derived access tokens (cf. 5.6.5.2.4).

The access token grants the right to use the specified object for the specified use to the bearer of the token.

The access token itself is a 256 bit value. It shall not be possible to predict the value of an access token. It shall also not be possible to guess the value of an access token with more than negligible probability. The access token shall provide a cryptographic strength of at least 128 bit. The internal structure of the access token is left to the specific implementation. Exchange of access tokens between SEs of different vendors is not required.

GTA API uses different types of access tokens to allow flexible definition of access policies (5.6.5.3).

5.6.5.2.2 Basic access tokens

To enable implementation of advanced access control between different applications, GTA API implementations can support basic access tokens. Basic access tokens enable implementing access control between different applications that can share a single SE (cf. 5.5.1.3). Basic access tokens are issued by the SE containing the respective objects.

Basic access tokens have no additional attributes. Basic access tokens can be retrieved from GTA API after system start-up until the function to retrieve access tokens is actively disabled. After the function is disabled, it will not become available again until the system (including the SE) is restarted. This behaviour is realized using a dedicated token issuing token (see 5.6.5.2.3).

5.6.5.2.3 Token issuing token

GTA API implementations supporting basic access tokens shall support granting of basic access tokens using a token issuing token. Basic access tokens can only be issued after authentication with a dedicated token issuing token. The token issuing token shall be provided only once during start-up of the device (cf. 5.6.6).

This procedure can be used to separate access between different applications. A privileged process run at system start-up gets hold of the token issuing token and requests basic access tokens for various objects that are protected by the SE. Either all required tokens are requested upfront and passed on to dependent processes or the privileged process provides a service to acquire access tokens. This privileged service should be protected by appropriate access control that is based on OS mechanism. The further distribution of the basic access tokens to subsequent applications is subject to the power of decision of the privileged process. Process privilege management and process isolation is the responsibility of the system OS.

5.6.5.2.4 Personality derived access tokens

GTA API implementations can support personality derived access tokens. Personality derived access tokens result from the successful use of a personality. Depending on the profile used, there are various possibilities for personality derived access tokens, for example:

- Dedicated personalities and profiles that are used to enable simple password or passcode-based authentication schemes (cf. ch.iec.30168.basic.passcode, Clause B.1).
- Dedicated personalities and profiles that are used for more complex authentication schemes, for example, challenge response based on symmetric or asymmetric secrets.
- Access tokens resulting from a side-effect of a general cryptographic protocol. For example, the successful verification of a digital signature or successful completion of a cryptographic key exchange can also yield an access token. This access token can then be used to authenticate further operations.

The attributes of a personality derived access token should include at least binding to the personality and the profile leading to the creation of the access token as illustrated in Figure 15. Other attributes can be used to further restrict the applicability of the access token. For example, the verification of a signature using a specific personality can result in an access token. The resulting access token then grants the privilege to set a trusted attribute of that personality to that exact value authenticated by the signature.

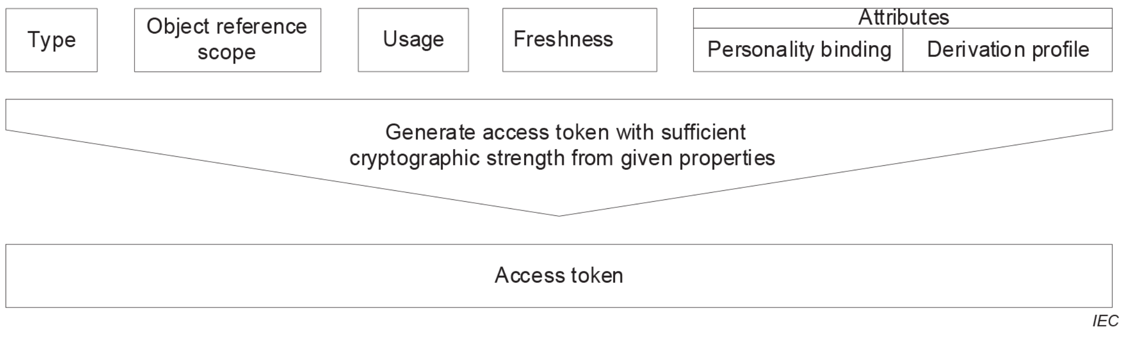


Figure 15 – Personality derived access token

It is possible to set up a chain of authentication protocols depending on different personalities to require multiple authentications before using a personality.

EXAMPLE A personality according to the profile ch.iec.30168.basic.passcode (Clause B.1) is used to create an access token resulting from the verification of a password. This access token is used to protect the usage of another personality according to the same profile ch.iec.30168.basic.passcode. The verification of the second password is only possible after providing the first password. The authentication token that is obtained upon verification of the second password is used to allow computation of an authentication artifact (for example, a digital signature). Thus, computation of the authentication artifact will only be possible after both passcodes are made available.

5.6.5.2.5 Physical presence access tokens

Physical presence access tokens can be acquired during device start-up. Physical presence tokens shall only be issued by GTA API if the device (and SE) has evidence that physical presence criteria are fulfilled (for example, keylock), cf. 5.6.6. Definition of these criteria is solution specific and out of scope of this document. Vendors shall describe these criteria for their respective solution.

The security impact that is achieved by privileges granted by physical presence shall not exceed the security impact achievable by an entity (potential attacker) exercising complete physical control over the device (for example, with a sledgehammer). Especially availability can be affected by privileges acquired by physical presence. For example, availability related privileges include deletion of secret or trusted attributes but not their creation or modification.

5.6.5.3 Access policies

Access policies are used to describe the rules which apply to allow the execution of a specific operation (cf. Table 1) on an object.

Access policies shall comply with the rules shown in Figure 16 in Backus–Naur form[3] (BNF).

[3] See, for example, https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

```
<Access Policy> ::=
          <Personality Access Policy>
        | <Device State Access Policy>
<Personality Access Policy> ::=
          <Initial Access>
        | <Advanced Access Policy>
<Advanced Access Policy> ::=
          <Basic Access Token Descriptor>
        | <Basic Access Token Descriptor> <Personality Derived Access>
        | <Personality Derived Access Token Descriptor> <Personality Derived Access>
<Personality Derived Access> ::=
        <Personality Derived Access Token Descriptor> <Personality Derived Access>
      | <Empty>
<Device State Access Policy> ::=
          <Physical Presence Access Token Descriptor> <Personality Derived Access>
        | <Personality Derived Access>
```

IEC

Figure 16 – Access policy composition (BNF)

That means an access policy for a personality requires initial access (cf. 5.6.5.1) or the presentation of a basic access token (cf. 5.6.5.2.2) or personality derived access token (5.6.5.2.4). There can be an optional choice of further personality derived access tokens. Access policies shall be evaluated as disjunctions ("or"). Access to a personality that is protected by a respective access policy is granted in case the application presents at least one access token according to the policy.

It is also possible to require multiple access tokens before an operation is granted ("and", conjunction). Conjunctions can be built by chaining personality derived access policies (cf. 5.6.5.2.4). For example, to require two distinct passcodes before usage of a personality holding an RSA private key, usage of the RSA key personality is protected by a first passcode personality. Verification of this first passcode is protected by a second passcode personality.

To fulfil the access policy for a device state, a physical presence access token or, optionally, a personality derived access token can be presented.

5.6.6 GTA API start-up

Granting of several access permissions depends on the start-up phase of the GTA API device (cf. Figure 17). Transition between GTA API start-up states is controlled by the runtime environment typically in relation to the boot phase of the device.

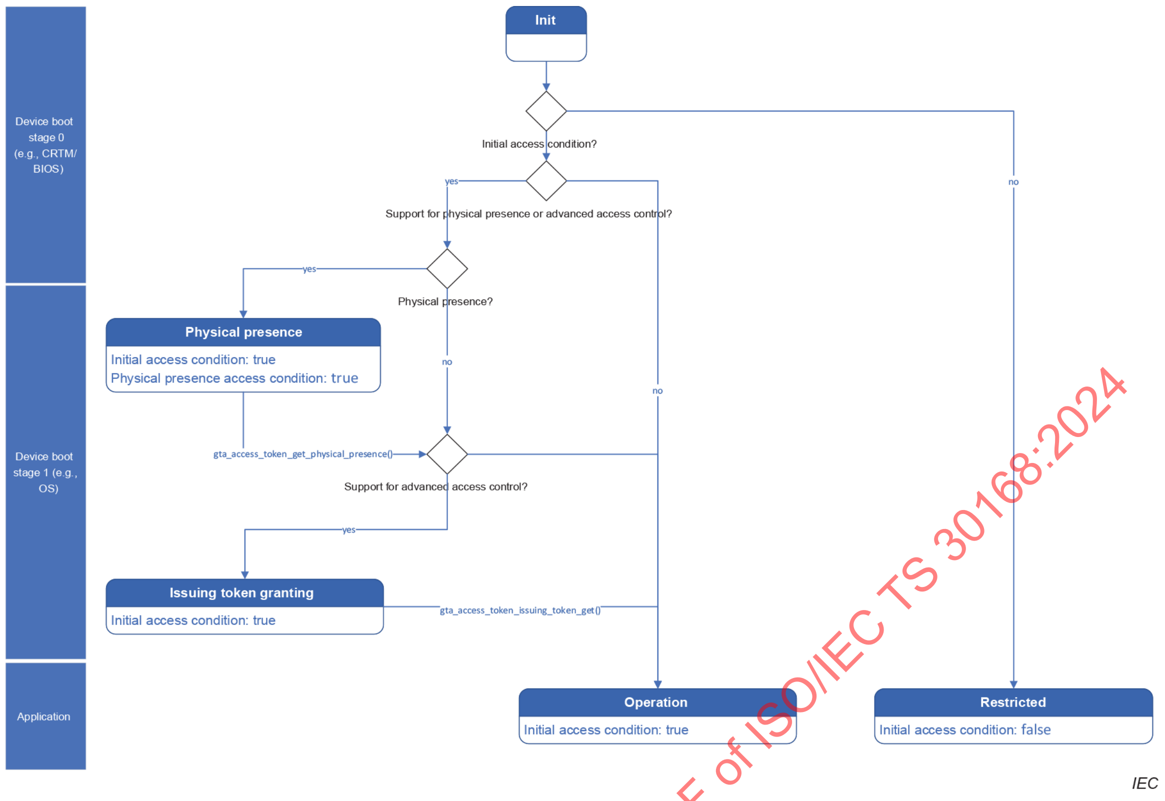

Device boot stage 0 (e.g., CRTM/ BIOS)
Device boot stage 1 (e.g., OS)
Application
Init
Initial access condition?
yes
no
Support for physical presence or advanced access control?
yes
no
Physical presence?
yes
no
Physical presence
Initial access condition: true
Physical presence access condition: true
gta_access_token_get_physical_presence()
Support for advanced access control?
yes
Issuing token granting
Initial access condition: true
gta_access_token_issuing_token_get()
Operation
Initial access condition: true
Restricted
Initial access condition: false
IEC

Figure 17 – GTA API start-up phases

GTA API recognizes the following start-up states:

- Init: This is the first state that is reached after system start-up. GTA API only provides limited functionality in this state. The runtime system decides whether the prerequisites to grant initial access and physical presence are met or not. The start-up phase of GTA API is advanced accordingly.
- Physical presence or no physical presence: Depending on a decision of the runtime environment, GTA API and the underlying SE either permit or reject operations requiring physical presence authentication (cf. 5.6.5.2.5). These start-up phases may be omitted in case the device does not support physical presence.
- Issuing token granting: This state is intended for a privileged process of the runtime system. The privileged process can get hold of the token issuing token (cf. 5.6.5.2.3) and collect basic access tokens (cf. 5.6.5.2.2). The basic access tokens are distributed by the privileged process among subsequent application processes. Application processes shall not be started before the token issuing token has been collected by the privileged process.
- Operation: This state covers the general usage of GTA API by applications.
- Restricted: This state covers the general usage of GTA API by applications with limited functionality if the initial access condition is not met. Especially it is not possible to use any functions which are subject to access control in this state.

The start-up phase state is system wide, that is, it is shared among all instances of GTA API running on a device. Resetting the GTA API start-up phase requires a power cycle of the device.

6 API specification

6.1 Overview

GTA API is structured into several function groups. Each function group addresses a specific aspect being relevant for different user types of GTA API. See Table 7.

Table 7 – GTA API function groups

Function group	Subclause	Description
General management functions	6.6.2	General functions to manage GTA API
Process synchronization	6.6.3	Functions to be used by GTA API for process synchronization. This interface shall be provided by the application. The platform vendor can provide appropriate functions to the application developer
Secure memory management	6.6.4	GTA API utility functions for secure memory management which can also be used by the application
Function parameter I/O streams	6.6.5	Parameters to and results from GTA API are passed using streams. The application developer can implement a stream interface according to the needs of the application. Platform vendors or providers of GTA API can provide reference implementations
Instance management functions	6.6.6	Functions to initialize and finalize GTA API instances to be used by the application
Context management functions	6.6.7	Functions to establish GTA API contexts that are used to organize the execution of GTA API functions
Access token functions	6.6.8	Functions to acquire access tokens required to authenticate for the use of privileged functions on protected GTA API objects
Device state management functions	6.6.9	Functions to manage the state of the device/SE
Identifier and personality management	6.6.10	Functions for the management (creation, deletion) of identifiers and personalities
Access policy management functions	6.6.11	Functions to create access policies to be attached to GTA API objects (for example, personalities)
Data protection functions	6.6.12	Functions to protect data
Channel protection functions	6.6.13	Functions to support the establishment of secure channels
Supplementary security functions	6.6.14	Security functions which do not relate to specific GTA API objects, for example, secure random numbers
Trusted execution environment	6.6.15	Functions to interface with the trusted execution environment which can optionally be provided by an SE
Secure element provider implementation support	6.6.16	Functions to support the implementor of SE providers to interface with the GTA API framework

Different implementations of GTA API can provide different extent of support for the individual functions and related features that are described in this document.

Table 8 provides an overview on the feature classes being distinguished. The description explains how an implementation of GTA API shall react in case a feature or functionality is not supported.

Table 8 – GTA API feature classes

Feature class	Description
T	Threading: Shall be supported in case GTA API is supposed to be used in parallel/multithreaded environments. Implementations not supporting parallel environments shall implement these functions as stubs. Calling these stub functions shall not result into an error condition or other side-effects.
U	Update: Dynamic update of the GTA API itself. In case the feature is not supported by the GTA API implementation, the functions shall fail with GTA_ERROR_FEATURE_NOT_SUPPORTED.
A	Advanced access control: Support of advanced access control. In case the feature is not supported by the GTA API implementation, the functions shall fail with GTA_ERROR_FEATURE_NOT_SUPPORTED.
X	Trusted eXecution: Support of trusted execution. In case the feature is not supported by the GTA API implementation, the functions shall fail with GTA_ERROR_FEATURE_NOT_SUPPORTED.
C	Create: This function can be omitted in case only the usage of static, preinitialized, and prepersonalized personalities is supported by the GTA API implementation. GTA API implementations which do not support this functionality shall fail with GTA_ERROR_FEATURE_NOT_SUPPORTED.
M	Mandatory: This function belongs to the mandatory subset of GTA API and shall be supported by all implementations.
P	Profile-specific: Supported depending on the used profile. The function shall behave according to the specification provided by the selected profile. In case the behaviour of the function is not defined by the respective profile, it shall fail with GTA_ERROR_PROFILE_UNSUPPORTED.

'C' (Create) takes precedence over 'P' (Profile-specific). That means GTA API implementations that do not support creation of personalities and attributes cannot support any profiles specifying these operations and shall fail with GTA_ERROR_FEATURE_NOT_SUPPORTED.

Table 9 specifies a feature class for each function of GTA API. Some of the features are optional. Independent of the feature class of a function, GTA API framework shall be able to address and exploit the full functionality of subordinate modules.

The function category indicates whether the behaviour of the function is inherent to the GTA API framework core, the host platform GTA API is running on, or the individual SE providers.

Table 9 – GTA API functions per feature class

Function	Feature class	Category
General management functions		
gta_library_info	M	Framework core
gta_register_provider	M	Framework core
gta_update_library	U	Framework core
Process synchronization		
gta_mutex_create	T	Host platform
gta_mutex_destroy	T	Host platform
gta_mutex_lock	T	Host platform
gta_mutex_unlock	T	Host platform
Secure memory management		
gta_secmem_calloc	M	Host platform
gta_secmen_checkptr	M	Host platform
gta_secmem_free	M	Host platform

Function	Feature class	Category
Instance management functions		
gta_instance_final	M	Framework core
gta_instance_init	M	Framework core
Context management functions		
gta_context_auth_get_challenge	P	SE provider
gta_context_auth_set_access_token	A	SE provider
gta_context_auth_set_random	P	SE provider
gta_context_close	M	Framework core
gta_context_open	M	Framework core
gta_context_get_attribute	P	SE provider
gta_context_set_attribute	P	SE provider
Access token functions		
gta_access_token_get_basic	A	SE provider
gta_access_token_get_pers_derived	P	SE provider
gta_access_token_get_issuing	A	SE provider
gta_access_token_get_physical_presence	A	SE provider
gta_access_token_revoke	A	SE provider
Device state management functions		
gta_devicestate_attestate	P	SE provider
gta_devicestate_recede	C	SE provider
gta_devicestate_transition	C	SE provider
Identifier and personality management		
gta_identifier_assign	C	SE provider
gta_identifier_enumerate	M	SE provider
gta_personality_add_attribute	C, P	SE provider
gta_personality_add_trusted_attribute	C, P	SE provider
gta_personality_attributes_enumerate	M	SE provider
gta_personality_attestate	P	SE provider
gta_personality_create	C	SE provider
gta_personality_deploy	C	SE provider
gta_personality_enroll	C, P	SE provider
gta_personality_enroll_auth	C, P	SE provider
gta_personality_enumerate	M	SE provider
gta_personality_enumerate_application	M	SE provider
gta_personality_get_attribute	P	SE provider
gta_personality_remove_attribute	P	SE provider
gta_personality_remove	C	SE provider
gta_personality_deactivate	P	SE provider
gta_personality_activate	P	SE provider
gta_personality_deactivate_attribute	P	SE provider
gta_personality_activate_attribute	P	SE provider

Function	Feature class	Category
Access policy management functions		
gta_access_policy_add_basic_access_token_descriptor	A	Framework core
gta_access_policy_add_pers_derived_access_token_descriptor	A	Framework core SE provider[a]
gta_access_policy_add_physical_presence_access_token_descriptor	A	Framework core SE provider[a]
gta_access_policy_create	A	Framework core
gta_access_policy_destroy	A	Framework core
gta_access_policy_get_access_descriptor_attribute	M	Framework core
gta_access_policy_get_access_descriptor_type	M	Framework core
gta_access_policy_simple	M	Framework core
gta_access_policy_enumerate	M	Framework core
Data protection functions		
gta_authenticate_data_detached	P	SE provider
gta_seal_data	P	SE provider
gta_unseal_data	P	SE provider
gta_verify_data_detached	P	SE provider
gta_verify	P	SE provider
Channel protection functions		
gta_seal _message	P	SE provider
gta_security_association_accept	P	SE provider
gta_security_association_destroy	P	SE provider
gta_security_association_initialize	P	SE provider
gta_unseal_message	P	SE provider
Supplementary security functions		
gta_get_random_bytes	M	SE provider
gta_attestate	P	SE provider
Trusted execution environment		
gta_trustex_function_install	X	SE provider
gta_trustex_function_uninstall	X	SE provider
gta_trustex_function_execute	X	SE provider
gta_trustex_function_terminate	X	SE provider
Secure element provider implementation support		
gta_provider_context_close	M	SE provider
gta_provider_context_open	M	SE provider
gta_provider_get_params	M	Framework core
gta_context_get_provider_params	M	Framework core
gta_context_get_params	M	Framework core

[a] If this function is used with GTA API, the used SE providers must support the created access policy. SE providers shall not permit access to objects protected by access policies not supported by the SE provider.

Annex F provides some additional guidance for the implementors of GTA API.

Annex G provides some examples for the usage of GTA API.

6.2 Language binding

Throughout this document C99 syntax is used to specify the API.

GTA API uses the following definitions of the C standard library:

- `stdbool.h`
- `stdint.h`
- `stddef.h`

It is possible that other language bindings will be specified in the future.

6.3 Endianness

Implementations shall use the endianness of the underlying host system to represent basic C types (int, char, ...). Whenever bytestrings are used to represent big numbers or other cryptographic artifacts, these bytestrings shall be in big endian. A profile (cf. 5.5.5) can explicitly require little endian encoding.

6.4 Exception handling

GTA API implementations shall be designed to never fail in a way that results in cryptographic compromise, for example:

- Wrong handling of total failure of random number generator
- Exposition of plaintext due to unwanted equality of input and output buffer
- Enablement of side-channel analysis through observation of exception handling

6.5 Using GTA API from an application

6.5.1 Header files

A typical C application imports API functionality by C header files. Each implementation of GTA API shall ensure that it can be interfaced with by an application using the header files given in Annex A. Implementations of GTA API may provide their own header files if these do not break compatibility with this document.

6.5.2 Call conventions and error handling

All functions that are defined within GTA API follow the following schema:

```
[bool|void *] gta_function([...,] gta_errinfo_t * p_errinfo);
```

The boolean interpretation of the return code tells whether the function completed successfully or not. If a function fails, the value that is referenced by p_errinfo can be used to obtain further information about the type of error. If a function completes successfully, the value that is referenced by p_errinfo is left unmodified by the GTA API function.

The following convention should be followed when calling a GTA API function:

```
gta_errinfo_t errinfo;
[void * p_res;]

if ([p_res = ] gta_function([...,] &errinfo)) {
  do_something();
}
else {
  do_error_handling(errinfo);
}
```

6.6 Types and function documentation

6.6.1 Basic types

6.6.1.1 Description

These are the basic types which are used throughout GTA API.

6.6.1.2 Defines documentation

6.6.1.2.1 GTA_ERROR_ACCESS

#define GTA_ERROR_ACCESS 15

Access to the function or object is not granted.

6.6.1.2.2 GTA_ERROR_ACCESS_POLICY

#define GTA_ERROR_ACCESS_POLICY 14

The access policy is invalid or cannot be used for this type of object.

6.6.1.2.3 GTA_ERROR_ATTRIBUTE_MISSING

#define GTA_ERROR_ATTRIBUTE_MISSING 13

A mandatory attribute for the function is missing.

6.6.1.2.4 GTA_ERROR_CONTEXT_BUSY

#define GTA_ERROR_CONTEXT_BUSY 16

The context cannot be used for the intended operation as another operation is still pending.

6.6.1.2.5 GTA_ERROR_ENUM_NO_MORE_ITEMS

#define GTA_ERROR_ENUM_NO_MORE_ITEMS 8

An enumeration went past the last item.

6.6.1.2.6 GTA_ERROR_FEATURE_NOT_SUPPORTED

#define GTA_ERROR_FEATURE_NOT_SUPPORTED 17

This optional feature is not supported by this implementation of GTA API.

6.6.1.2.7 GTA_ERROR_GENERIC_DEVICE_ERROR

#define GTA_ERROR_GENERIC_DEVICE_ERROR -1

Negative error codes are reserved for device specific errors within the SE provider.

6.6.1.2.8 GTA_ERROR_HANDLE_INVALID

#define GTA_ERROR_HANDLE_INVALID 2

The handle is invalid.

6.6.1.2.9 GTA_ERROR_HANDLES_EXAUSTED

#define GTA_ERROR_HANDLES_EXAUSTED 4

The maximum amount of handles has been exhausted.

6.6.1.2.10 GTA_ERROR_INTERNAL_ERROR

#define GTA_ERROR_INTERNAL_ERROR 1

A generic internal error has occurred.

6.6.1.2.11 GTA_ERROR_INVALID_ATTRIBUTE

#define GTA_ERROR_INVALID_ATTRIBUTE 12

The specified attribute is invalid.

6.6.1.2.12 GTA_ERROR_INVALID_PARAMETER

#define GTA_ERROR_INVALID_PARAMETER 7

Some function parameter is wrong.

6.6.1.2.13 GTA_ERROR_ITEM_NOT_FOUND

#define GTA_ERROR_ITEM_NOT_FOUND 10

An object (identifier, personality, attribute) with the specified name does not exist.

6.6.1.2.14 GTA_ERROR_MEMORY

#define GTA_ERROR_MEMORY 5

There is not enough memory to complete the function.

6.6.1.2.15 GTA_ERROR_NAME_ALREADY_EXISTS

#define GTA_ERROR_NAME_ALREADY_EXISTS 9

An attempt to create an object or attribute with an already existing name failed.

6.6.1.2.16 GTA_ERROR_PROFILE_UNSUPPORTED

#define GTA_ERROR_PROFILE_UNSUPPORTED 11

The requested profile is not supported for this operation.

6.6.1.2.17 GTA_ERROR_PROVIDER_INVALID

#define GTA_ERROR_PROVIDER_INVALID 6

The secure element provider is invalid.

6.6.1.2.18 GTA_ERROR_PTR_INVALID

#define GTA_ERROR_PTR_INVALID 3

An invalid pointer was used.

6.6.1.2.19 GTA_ERROR_STREAM_EOF

#define GTA_ERROR_STREAM_EOF 20

No more data is available from the stream.

6.6.1.2.20 GTA_HANDLE_ENUM_FIRST

#define GTA_HANDLE_ENUM_FIRST ((gta_context_handle_t)-1)

Indicator value used to start an enumeration operation.

6.6.1.2.21 GTA_HANDLE_INVALID

#define GTA_HANDLE_INVALID ((gta_context_handle_t) 0)

Value used for invalid handles.

6.6.1.3 Typedef documentation

6.6.1.3.1 gta_errinfo_t

typedef long gta_errinfo_t;

GTA API error information. Detailed error information is given as signed long integer. Positive error codes shall only use errors that are defined by the GTA API. Negative error codes are reserved to indicate device errors specific to the implementation of an SE. Interpretations of error codes applicable to specific functions are explained within the respective function descriptions.

6.6.1.3.2 gta_enum_handle_t

typedef gta_handle_t gta_enum_handle_t;

Enumeration handle

6.6.1.3.3 gta_handle_t

*typedef struct gta_handle * gta_handle_t;*

Generic handle that is used to hide implementation details (opaque struct)

6.6.1.3.4 gta_profile_name_t

*typedef char * gta_profile_name_t;*

Profile name identification for a profile. Zero terminated string in UTF8 coding.

To avoid collisions, profiles rely on a hierarchical namespacing scheme that is derived from reversed Internet domain name identifiers.

EXAMPLE ch.iec.30168.* for any profiles defined directly within ISO/IEC TS 30168.

Profiles are either hardcoded or configured into the GTA API implementation and are not intended to be changed by the application.

Apart from application-specific profiles to be defined outside this document, every implementation of GTA API shall support the basic profiles that are defined in Table 10 according to the detailed description of the basic profiles given in Annex B.

Table 10 – Basic profiles

Profile	Functions	Description
ch.iec.30168.basic.passcode	**gta_personality_deploy()**, **gta_verify()**, **gta_access_token_get_pers_derived()**	Verify a passcode and acquire an access token in return.
ch.iec.30168.basic.local_data_integrity_only	**gta_personality_create()**, **gta_seal_data()**, **gta_unseal_data()**, **gta_authenticate _data_detached()**, **gta_verify_data_detached()**	Integrity protection for device local data; The integrity of the data is tied to the current device. An attempt to verify the data on a different device or to verify data that was sealed by a different device shall fail.
ch.iec.30168.basic.local_data_protection	**gta_personality_create()**, **gta_seal_data()**, **gta_unseal_data**	Integrity and confidentiality protection, for example, an authenticated encryption scheme; The integrity and confidentiality of the data are tied to the current device. An attempt to verify or decrypt the data on a different device or data that were sealed by a different device shall fail.

6.6.2 General management functions

6.6.2.1 Description

These are functions for the general management of GTA API. These functions include:

- Version management
- Update handling
- SE provider registration and management

6.6.2.2 Structure Documentation

6.6.2.2.1 gta_function_list_t

```
#define GTA_FUNCTION_INFO(return_type, function_name, argument_list) \
    GTA_DECLARE_FUNCTION_PTR(return_type, pf_##function_name##_t, argument_list)
#include "gta_apif.h"



struct gta_function_list_t {
    #define GTA_FUNCTION_INFO(return_type, function_name, argument_list) \
      pf_##function_name##_t pf_##function_name
};
```

List of pointers to functions that shall be implemented by an SE provider.

Apart from **gta_context_open()** and **gta_context_close()** all function signatures are unchanged between the GTA API application interface and the provider interface.

gta_context_open() and **gta_context_close()** use the dedicated provider notification functions **gta_provider_context_open()** and **gta_provider_context_close()**. The reason for this deviation is that the context object should be managed by the GTA API framework, and not by the individual provider. The framework passes a preinitialized context to the provider, which allows the provider to access GTA API framework functions.

6.6.2.2.2 gta_info_t

```
struct gta_info_t {
    long ts_version;
    long ts_abi_compat_version;
    long library_version;
    long max_contexts;
};
```

GTA API library information.

Member Data Documentation:

Type	Name	Description
long	*ts_version*	Version of ISOIEC TS 30168 that applies. Implementations using this version of ISO/IEC TS 30168 shall set *ts_version* to 1.
long	*ts_abi_compat_version*	Minimum ABI version to which this implementation of GTA API is compatible. For example, a library with *ts_version* = 3 and *ts_abi_compat_version* = 1 is compatible to an application developed for *ts_version* = 1.
long	*library_version*	Vendor implementation specific version information; Implementations with higher library versions shall provide at least the functionality of previous versions.
long	*max_contexts*	Maximum amount of GTA API contexts that can be opened in parallel.

6.6.2.2.3 gta_provider_info_t

```
struct gta_provider_info_t {
    uint32_t                        version;
    gta_provider_info_type_t        type;
    gta_provider_init_t             provider_init;
    gtaio_istream_t *               provider_init_config;
    struct {
        gta_profile_name_t                      profile_name;
        struct gta_protection_properties_t      protection_properties;
        uint8_t                                 priority;
    } profile_info;
};
```

Information on a specific SE provider to be registered and used by GTA API.

Member Data Documentation:

Type	Name	Description
uint32_t	*version*	Version information of the provider
gta_provider_info_type_t	*type*	Provider type and registration method; Only GTA_PROVIDER_INFO_CALLBACK is supported for this version of GTA API.
gta_provider_init_t	*provider_init*	Initialization callback to be used by GTA API to initialize the provider.
gtaio_istream_t *	*provider_init_config*	Extra configuration for the provider; This information is forwarded to the initialization callback.
gta_profile_name_t	*profile_info.profile_name*	Name of the supported profile.
gta_protection_properties_t	*profile_info.profile_name*	Protection properties that are provided by the SE used by the provider
uint8_t	*profile_info.priority*	Prioritization of the provider that is used in case multiple providers are eligible for the same profile. Providers with lower values are preferred.

6.6.2.3 Typedef documentation

6.6.2.3.1 gta_provider_init_t

```
typedef const struct gta_function_list_t*(* gta_provider_init_t) (
    gta_context_handle_t h_ctx,
    gtaio_istream_t * provider_init_config,
    gtaio_ostream_t * logging,
    void **pp_params,
    void (** ppf_free_params)(void * p_params),
    gta_errinfo_t *p_errinfo);
```

Initialize an SE provider.

This callback function is called from within **gta_register_provider()** to allow the provider to run its own initialization code.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle for a GTA API context that can be used by the provider to call general framework functions (for example, gta_secmem_). The context is automatically provided by **gta_register_provider()**. Resources that are associated with this context are kept until the provider is finalized.
in	*provider_init_config*	Configuration for the provider; The information that is provided by the provider_init_config stream depends on the respective provider. The stream can be used to pass any type of vendor defined information for provider initialization.
in	*logging*	Stream that can be used by the provider to write log messages. In case no log messages are required this parameter is set to NULL.
out	*pp_params*	Pointer to accept a reference to provider local parameters. Any function that is implemented by the provider can access this reference using **gta_provider_get_params()** or **gta_context_get_provider_params()**.
out	*ppf_free_params*	Pointer to accept a reference to a callback function to be called from **gta_instance_final()** to cleanup provider local parameters.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Pointer to the provider function list (see **gta_function_list_t**) or NULL if there is a failure

6.6.2.4 Enumeration type documentation

6.6.2.4.1 enum gta_provider_info_type_t

typedef enum {

GTA_PROVIDER_INFO_CALLBACK = 0

} gta_provider_info_type_t;

Type of SE provider and provider registration method.

Enumerator:

Enum	Description
GTA_PROVIDER_INFO_CALLBACK	The SE provider is registered and initialized using an application provided initialization callback (see **gta_provider_init_t()**).

6.6.2.4.2 gta_library_info

bool gta_library_info (

*struct gta_info_t * p_gta_info,*

*gta_errinfo_t * p_errinfo)*

Query information about the GTA API library.

Parameters:

in/out	Name	Description
out	*p_gta_info*	Library information is written to the referenced struct.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.2.4.3 gta_register_provider

bool gta_register_provider (

gta_instance_handle_t h_inst,

*const struct gta_provider_info_t * p_provider_info,*

*gta_errinfo_t * p_errinfo)*

Register an SE provider for use with a specific profile.

gta_register_provider() is called by an application to register an SE provider with the SE provider interface of the GTA API framework. Typically, an application will register all required providers/profiles at start-up. It is up to the application to manage available providers, for example, if a new SE is added or updated. If a single SE provider supports multiple profiles, the function must be called multiple times – once for every profile.

Parameters:

in/out	Name	Description
in	*h_inst*	Handle to the GTA API instance (see **gta_instance_init()**)
in	*p_provider_info*	Information on the provider to be registered (see **gta_provider_info_t**)
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.2.4.4 gta_update_library

bool gta_update_library (

*gtaio_istream_t * update_stream,*

*gta_errinfo_t * p_errinfo*

Update the GTA API implementation or subcomponents thereof, or both.

This function is used to provide an update for the GTA API library implementation or the underlying SE. This function shall never be started while any contexts are opened. After this function has returned successfully, any subsequent calls are expected to use the updated functionality.

NOTE It is left to the implementor to decide which parts of the GTA API are updateable using this function. In case a hardware SE is used, this function is the preferred way to provide firmware updates for the SE.

Parameters:

in/out	Name	Description
in	*update_stream*	BLOB data stream containing the update information.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.3 Process synchronization

6.6.3.1 Description

This group of functions allows GTA API to interface with the host platform for synchronization (mutexes) and memory management.

6.6.3.2 Structure documentation

6.6.3.2.1 gta_os_functions_t

```
struct gta_os_functions_t {
    /* Memory management */
    calloc_t calloc;
    free_t free;


    /* Synchronization functions */
    mutex_create_t mutex_create;
    mutex_destroy_t mutex_destroy;
    mutex_lock_t mutex_lock;
    mutex_unlock_t mutex_unlock;
};
```

Host platform-specific functions to be used by the GTA API implementation.

- Application provided memory management callback functions
- Synchronization function call-backs passed from the application to allow GTA API to synchronize internal access to shared resources. These function pointers shall not be directly used from within the GTA API implementation. Instead access to these functions shall be redirected through **gta_mutex_create()**, **gta_mutex_destroy()**, **gta_mutex_lock()**, and **gta_mutex_unlock()**. These functions are supposed to realize the respective functionality based on the application callbacks.

Member Data Documentation:

Type	Name	Description
calloc_t	*calloc*	see 6.6.3.3.1
free_t	*free*	see 6.6.3.3.2
mutex_create_t	*mutex_create*	see 6.6.3.3.4
mutex_destroy_t	*mutex_destroy*	see 6.6.3.3.5
mutex_lock_t	*mutex_lock*	see 6.6.3.3.6
mutex_unlock_t	*mutex_unlock*	see 6.6.3.3.7

6.6.3.3 Typedef documentation

6.6.3.3.1 calloc_t

typedef void(* calloc_t) (*

size_t n,

size_t size);

Allocate a continuous block of dynamic memory. The content of the new memory block shall be set to 0.

Parameters:

in/out	Name	Description
in	*n*	number of elements
in	*size*	size of each element

Returns:

Pointer to the allocated memory block.

6.6.3.3.2 free_t

typedef void(free_t) (*

*void *ptr);*

Free a memory block previously allocated with **calloc_t**.

Parameters:

in/out	Name	Description
in	*ptr*	Pointer to the memory block

6.6.3.3.3 gta_mutex_t

typedef void gta_mutex_t;*

GTA API mutex type

6.6.3.3.4 mutex_create_t

typedef gta_mutex_t (mutex_create_t) ();*

Create a mutex object.

Returns:

Reference to new mutex object

6.6.3.3.5 mutex_destroy_t

typedef bool(mutex_destroy_t) (gta_mutex_t mutex);*

Destroy a mutex object.

Parameters:

in/out	**Name**	**Description**
in	*mutex*	Reference to the mutex object

Returns:

True on success, false on failure

6.6.3.3.6 mutex_lock_t

typedef bool(mutex_lock_t) (gta_mutex_t mutex);*

Lock a mutex object.

Parameters:

in/out	**Name**	**Description**
in	*mutex*	Reference to the mutex object

Returns:

True on success, false on failure

6.6.3.3.7 mutex_unlock_t

typedef bool(mutex_unlock_t) (gta_mutex_t mutex);*

Unlock mutex.

Parameters:

in/out	**Name**	**Description**
in	*mutex*	Reference to the mutex object

Returns:

True on success, false on failure

6.6.3.4 Function documentation

6.6.3.4.1 gta_mutex_create

gta_mutex_t gta_mutex_create (gta_context_handle_t h_ctx)

Create a mutex object.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**

Returns:

Reference to the new mutex object.

6.6.3.4.2 gta_mutex_destroy

bool gta_mutex_destroy (gta_context_handle_t h_ctx, gta_mutex_t mutex)

Destroy a mutex object.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*mutex*	Reference to the mutex object

Returns:

True on success, false on failure

6.6.3.4.3 gta_mutex_lock

bool gta_mutex_lock (gta_context_handle_t h_ctx, gta_mutex_t mutex)

Lock mutex (blocking).

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*mutex*	Reference to the mutex object

Returns:

True on success, false on failure

6.6.3.4.4 gta_mutex_unlock

bool gta_mutex_unlock (gta_context_handle_t h_ctx, gta_mutex_t mutex)

Unlock mutex.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*mutex*	Reference to the mutex object

Returns:

True on success, false on failure

6.6.4 Secure memory management

6.6.4.1 Description

Both applications and GTA API itself (especially implementations for SE providers) are potentially required to manage sensitive information within their process memory space. GTA API provides functions that can be used for secure memory management and memory resource tracking offering the following benefits:

- Memory buffers are guaranteed to be set to 0 when allocated using **gta_secmem_calloc()**.
- Memory buffers are guaranteed to be set to 0 after being freed using **gta_secmem_free()**.
- It is possible to check whether a pointer refers to a valid memory location, that is the location has been allocated using **gta_secmem_calloc()**.
- Memory space that is allocated is tracked within a GTA API context (see **gta_context_open()**) and automatically freed when the respective context is closed (see **gta_context_close()**).

Typically, GTA API relies on the application provided callback functions for memory management (see **gta_os_functions_t**).

6.6.4.2 Function documentation

6.6.4.2.1 gta_secmem_calloc

void gta_secmem_calloc (*

gta_context_handle_t h_ctx,

size_t n, size_t size,

*gta_errinfo_t * p_errinfo)*

Allocate a piece of memory that can be released using **gta_secmem_free()**.

This function should be implemented using the host-specific function for memory allocation that is passed within **gta_os_functions_t**. The implementation shall retain the information that is required to securely free the piece of memory transparent for the application. In case the context used to allocate the piece of memory is closed using **gta_context_close()**, the allocated memory is securely freed up.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*n*	Number of elements
in	*size*	Size of each element
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Pointer to the allocated memory array or NULL if there is a failure

6.6.4.2.2 gta_secmem_checkptr

void gta_secmem_checkptr (*

gta_context_handle_t h_ctx,

*void * p_check,*

*gta_errinfo_t * p_errinfo)*

Check whether a pointer refers to a valid memory location.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*p_check*	Pointer to be checked
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Value of p_check or NULL if there is a failure

6.6.4.2.3 gta_secmem_free

bool gta_secmem_free (

gta_context_handle_t h_ctx,

*void * ptr,*

*gta_errinfo_t * p_errinfo)*

Release a piece of memory that has been allocated by **gta_secmem_calloc()**.

The memory is released and cleared. Future access to the referenced memory location shall not allow recovery of any data previously stored at that location.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*ptr*	Pointer to the memory to be released
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.5 Function parameter I/O streams

6.6.5.1 Description

Streams are the mechanism that is used to pass data (for example, plaintext) to GTA API functions and to receive data (for example, ciphertext) from GTA API functions. Memory and buffer management is hidden within the stream implementation. GTA API only defines the functions that are required by the GTA API provider functions to receive and return data. GTA API provider functions read input data from input streams (gtaio_istream_t) and write return data to output streams (gtaio_ostream_t). How data is read or written to a stream by the application is out of scope.

The stream-based approach provides the following benefits:

- Resource management (for example, memory) can be implemented by the application developer however it fits best.
- Avoidance of unnecessary memory copy operations, for example, by passing data through the API in small chunks which can be transferred to a larger storage whenever a call completes.
- Different types of streams (buffer, file, socket) can be implemented. The interfaces shall comply with the definition for gtaio_istream_t and gtaio_ostream_t.
- Checking for potential overflows can be implemented at a single point within the stream implementation. It can be performed automatically whenever data is read or written to the stream by the API implementation.
- Streams allow asynchronous operation between GTA API and application.

The following example code illustrates how data can be received and returned using the stream interface. The example code returns a bytewise copy of data received from an input stream to an output stream:

```
void some_function(gtaio_istream_t * istream, gtaio_ostream_t * ostream)
{
    gta_errinfo_t errinfo = 0;
    bool b_rwloop = true;
    unsigned char byte_buf = 0;

    while (    b_rwloop
           && !istream->eof(istream,
                            &errinfo))
    {
        if (   istream->read(istream,
                             byte_buf,
                             1,
                             &errinfo)
               != 1
            || ostream->write(ostream,
                              &byte_buf,
                                  1,
                                  &errinfo)
                 != 1)
         {
             b_rwloop = false;
         }
     }
     ostream->finish(ostream, errinfo);
}
```

Code using the stream interface shall be prepared to handle both blocking and non-blocking behaviour of input and output streams.

As long as the method gtaio_istream.eof() does not return true, it is possible that the function will block until additional data is available. Likewise, it is possible that gtaio_ostream.write() will block to wait until the stream is able to accept the data.

Clause G.5 provides an example for the implementation of gtaio_istream_t and gtaio_ostream_t to pass simple parameters using memory buffers.

6.6.5.2 Structure documentation

6.6.5.2.1 gtaio_istream

typedef struct gtaio_istream gtaio_istream_t;

struct gtaio_istream {

gtaio_stream_read_t read;

gtaio_stream_eof_t eof;

*void * p_reserved2; /* write */*

*void * p_reserved3; /* finish */*

/ … */*

};

gtaio_istream is used to pass data from an application into GTA API. gtaio_istream defines an abstract interface (opaque struct) that must be extended to provide concrete istream implementations.

Member Data Documentation:

Type	Name	Description
gtaio_stream_read_t	*read*	method read, see 6.6.5.3.3
gtaio_stream_eof_t	*eof*	method eof, see 6.6.5.3.1
void *	*p_reserved2*	
void *	*p_reserved3*	

6.6.5.2.2 gtaio_ostream

typedef struct gtaio_istream gtaio_ostream_t;

struct gtaio_ostream {

*void * p_reserved0; /* write */*

*void * p_reserved1; /* eof */*

```
    gtaio_stream_write_t write;
    gtaio_stream_finish_t finish;
    /* … */
};
```

gtaio_ostream is used to return data from GTA API to the calling application. gtaio_ostream defines an abstract interface (opaque struct) that must be extended to provide concrete ostream implementations.

Member Data Documentation:

Type	Name	Description
void *	*p_reserved0*	
void *	*p_reserved1*	
gtaio_stream_write_t	*write*	method write, see 6.6.5.3.4
gtaio_stream_finish_t	*finish*	method eof, see 6.6.5.3.2

6.6.5.3 Typedef documentation

6.6.5.3.1 gtaio_stream_eof_t

```
typedef bool(* gtaio_stream_eof_t) (
    gtaio_istream_t *istream,
    gta_errinfo_t *p_errinfo)
```

Signal eof of an istream.

Parameters:

in/out	Name	Description
in	*istream*	the stream
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True in case EOF has been reached.

6.6.5.3.2 gtaio_stream_finish_t

```
typedef bool(* gtaio_stream_finish_t) (
    gtaio_ostream_t *ostream,
    gta_errinfo_t errinfo
    gta_errinfo_t *p_errinfo)
```

Tell an ostream that the GTA API function has finished writing to the stream.

By convention each GTA API function shall call this function after its last call to the **gtaio_ostream_t.write()** method.

Parameters:

in/out	Name	Description
in	*ostream*	The stream
in	*errinfo*	Final error condition of the GTA API function writing its result to the stream; Successful completion results in *errinfo* being set to 0. The error information provided to **gtaio_stream_finish_t** shall indicate the same error condition as the error information returned by the rGTA API function by its *errinfo_t * p_errinfo* return parameter.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.5.3.3 gtaio_stream_read_t

typedef size_t(gtaio_stream_read_t) (*
*gtaio_istream_t *istream,*
*char *data, size_t len,*
*gta_errinfo_t *p_errinfo)*

Read bytes from an istream.

Parameters:

in/out	Name	Description
in	*istream*	The stream
out	*data*	Buffer to receive the data read from the stream
in	*len*	Amount of data requested from the stream
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Amount of data returned from the stream.

In case less data than requested has been read, further information is available from the value that is referenced by p_errinfo.

Error Information	Description
GTA_ERROR_STREAM_EOF	All available data has been returned from the stream.

6.6.5.3.4 gtaio_stream_write_t

typedef size_t(gtaio_stream_write_t) (*

*gtaio_ostream_t *ostream,*

*const char *data,*

size_t len,

*gta_errinfo_t *p_errinfo)*

Write bytes to an ostream.

Parameters:

in/out	Name	Description
in	*ostream*	The stream
in	*data*	Data to write
in	*len*	Length of data to write
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Amount of data that was successfully written to the stream.

In case less data than requested has been written, further information is available from the value that is referenced by p_errinfo.

6.6.6 Instance management functions

6.6.6.1 Description

Instances allow each application to maintain independent views on GTA API. Before an application can use functions from GTA API, the application must set up an instance of the API. This setup is done using **gta_instance_init()**. Upon initialization, the application provides initialization parameters which allow the GTA API implementation to interface with the application and the host platform.

6.6.6.2 Structure Documentation

6.6.6.2.1 gta_instance_params_t

struct gta_instance_params_t {

gta_mutex_t global_mutex;

struct gta_os_functions_t os_functions;

*gtaio_ostream_t * logging;*

};

Parameters to setup a new instance of GTA API used with **gta_instance_init()**.

Member Data Documentation:

Type	**Name**	**Description**
gta_mutex_t	*global_mutex*	Pointer to an application global mutex reserved for GTA API; In case the application process wants to access the GTA API from multiple threads in parallel, this pointer shall refer to a valid mutex. The application must initialize this mutex with **os_functions.create_mutex()** prior to calling **gta_context_open()**. All calls to **gta_instance_init()** from within the same process shall reference the same global mutex. *global_mutex* can be used to manage globally shared resources within the GTA API implementation. In case no multi-threaded access is required, *global_mutex* is set to NULL. In case *global_mutex* is not set to NULL, the synchronization functions **os_functions.create_mutex**, **os_functions.lock_mutex**, **os_functions.unlock_mutex**, and **os_functions.destroy_mutex** shall be provided as part of *os_functions.*
gta_os_functions_t	*os_functions*	Host platform-specific functions to be used by the GTA API implementation
gtaio_ostream_t *	*logging*	Stream that can be used by the GTA API instance to write log messages; In case no log messages are expected, it can be set to NULL.

6.6.6.3 Typedef documentation

6.6.6.3.1 gta_instance_handle_t

typedef gta_handle_t gta_instance_handle_t

Instance handle

6.6.6.4 Function documentation

6.6.6.4.1 gta_instance_final

*bool gta_instance_final (gta_instance_handle_t h_inst, gta_errinfo_t * p_errinfo)*

Finalize a GTA API instance.

This function closes a library instance that was opened with **gta_context_open()**. All resources that are associated with the instance are released.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.6.4.2 gta_instance_init

gta_instance_handle_t gta_instance_init (

*const struct gta_instance_params_t * p_instance_params,*

*gta_errinfo_t * p_errinfo)*

Open a new GTA API instance.

Instances are used to keep different applications apart. All instances view the same static objects, such as personalities or trust lists. Dynamic information like authentication states or session credentials are not shared between instances.

Parameters:

in/out	Name	Description
in	*p_instance_params*	Setup parameters for the new instance
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Handle to the new instance or **GTA_HANDLE_INVALID** on failure.

6.6.7 Context management functions

6.6.7.1 Description

Contexts are used to organize operations involving a personality according to a specific profile. Contexts track the authentication required to run privileged functions. Several GTA API functions can only be started after a matching access token has been provided to the context used to run the function.

6.6.7.2 Typedef documentation

6.6.7.2.1 gta_context_attribute_type_t

typedef char gta_context_attribute_type_t*

Context attribute type

6.6.7.2.2 gta_context_handle_t

typedef gta_handle_t gta_context_handle_t

Context handle

6.6.7.3 Function documentation

6.6.7.3.1 gta_context_auth_get_challenge

bool gta_context_auth_get_challenge (

gta_context_handle_t h_ctx,

*gtaio_ostream_t * challenge,*

*gta_errinfo_t * p_errinfo)*

Get a challenge that is required in case a challenge/response protocol is used to compute a personality derived access token.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
out	*challenge*	Challenge provided to the partner to perform a cryptographic protocol
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.7.3.2 gta_context_auth_set_access_token

bool gta_context_auth_set_access_token (

gta_context_handle_t h_ctx,

const gta_access_token_t access_token,

*gta_errinfo_t * p_errinfo)*

Provide an access token for using privileged functions with the context.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*access_token*	Access token that is provided to the context
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.7.3.3 gta_context_auth_set_random

bool gta_context_auth_set_random (

gta_context_handle_t h_ctx,

*gtaio_istream_t * random,*

*gta_errinfo_t * p_errinfo)*

Set an externally provided random that is required in case a challenge/response protocol is used to compute a personality derived access token.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*random*	Random provided by a partner to perform a cryptographic protocol
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.7.3.4 gta_context_close

bool gta_context_close (

gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Close an existing context.

This function closes a personality context that was opened with **gta_context_open()**. All resources that are associated with the context are released.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.7.3.5 gta_context_open

gta_context_handle_t gta_context_open (

gta_instance_handle_t h_inst,

const gta_personality_name_t personality,

const gta_profile_name_t profile,

*gta_errinfo_t * p_errinfo)*

Open a new context.

Contexts are used to organize coherent/joint operations relating to a specific personality, for example, a key agreement, data protection/unprotection. Operations running within the same context can share information, for example, a subsequent operation can be based on previous results.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*personality*	Personality to use for the context
in	*profile*	Profile to use for the context
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Handle to the new context or **GTA_HANDLE_INVALID** on failure.

6.6.7.3.6 gta_context_get_attribute

bool gta_context_get_attribute (

gta_context_handle_t h_ctx,

const gta_context_attribute_type_t attrtype,

*gtaio_ostream_t * p_attrvalue,*

*gta_errinfo_t * p_errinfo)*

Get an attribute resulting as a side-effect from an operation performed within the specified context.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrtype*	Type of attribute to be retrieved
out	*p_attrvalue*	Value of the requested attribute
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.7.3.7 gta_context_set_attribute

bool gta_context_set_attribute (

gta_context_handle_t h_ctx,

const gta_context_attribute_type_t attrtype,

*gtaio_istream_t * p_attrvalue,*

*gta_errinfo_t * p_errinfo)*

Set an attribute to be used with an operation performed within the specified context.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrtype*	Type of attribute to be set
in	*p_attrvalue*	Value for the attribute
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.8 Access token functions

6.6.8.1 Description

Access tokens are used to acquire privileges for the execution of GTA API functions.

Table 11 gives an overview of functions which may require an access token to be set before they can be run (see 5.6.5). The type of access token depends on the access policy which has been defined for the underlying information object, that is, personality or device state.

Table 11 – GTA API functions with access control

Function	Usage		
	Use	Admin	Recede
gta_devicestate_attestate	X		
gta_devicestate_recede			X
gta_personality_add_trusted_attribute		X	
gta_personality_attestate	X		
gta_personality_enroll	X		
gta_personality_enroll_auth	X		
gta_personality_remove_attribute		X	
gta_personality_activate		X	
gta_personality_deactivate_attribute		X	
gta_personality_activate_attribute		X	
gta_authenticate_data_detached	X		
gta_seal_data	X		
gta_unseal_data	X		
gta_verify_data_detached	X		
gta_verify	X		
gta_security_association_accept	X		
gta_security_association_initialize	X		

The following functions are used to manage and acquire access tokens.

6.6.8.2 Defines documentation

6.6.8.2.1 GTA_ACCESS_TOKEN_LEN

#define GTA_ACCESS_TOKEN_LEN (256/8)

Access token length

6.6.8.3 Typedef documentation

6.6.8.3.1 gta_access_token_t

typedef char gta_access_token_t[GTA_ACCESS_TOKEN_LEN]

Access token value

6.6.8.4 Function documentation

6.6.8.4.1 gta_access_token_get_basic

```
bool gta_access_token_get_basic (
    gta_instance_handle_t h_inst,
    const gta_access_token_t granting_token,
    const gta_personality_name_t personality_name,
    gta_access_token_usage_t usage,
    gta_access_token_t basic_access_token,
    gta_errinfo_t * p_errinfo)
```

Get a basic access token enabling access to privileged operations with personalities and device states.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle. (see **gta_instance_init()**)
in	*granting_token*	Token granting the privilege to issue additional basic access tokens. (see **gta_access_token_get_issuing()**)
in	*personality_name*	Name of the personality for which this token shall be valid. In case usage is set to GTA_USAGE_RECEDE, this parameter is ignored and should be set to NULL.
in	*usage*	Type of usage granted by the token
out	*basic_access_token*	Issued basic access token
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_FEATURE_NOT_SUPPORTED	This implementation of GTA API does not support basic access tokens.

6.6.8.4.2 gta_access_token_get_pers_derived

bool gta_access_token_get_pers_derived (

gta_context_handle_t h_ctx,

const gta_personality_name_t target_personality_name,

gta_access_token_usage_t usage,

*gta_access_token_t * p_pers_derived_access_token,*

*gta_errinfo_t * p_errinfo)*

Get a personality derived access token enabling access to privileged operations with personalities and device states.

This function can be used to derive an access token depending on the state of the provided context. The state that is required for successful derivation of an access token typically depends on the result of a previous operation using the personality associated to the context, for example:

- An explicit authentication operation, for example, performing a challenge/response protocol
- An implicit authentication operation, for example, resulting from the successful public key signature verification operation during some cryptographic protocol

The type of protocols and conditions that are required to derive an access token from the context are defined by the profile associated to the context.

The functions **gta_context_auth_get_challenge()** and **gta_context_auth_set_random()** can be used to support the implementation of authentication protocols provided by an SE provider.

Parameters:

in/out	**Name**	**Description**
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*target_personality_name*	Name of the personality for which this token shall be valid. In case usage is set to GTA_USAGE_RECEDE, this parameter is ignored and should be set to NULL.
in	*usage*	Type of usage granted by the token
out	*p_pers_derived_access_token*	Value of the issued personality derived access token
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	**Description**
GTA_ERROR_FEATURE_NOT_SUPPORTED	This implementation of GTA API does not support personality derived access tokens.

6.6.8.4.3 gta_access_token_get_issuing

bool gta_access_token_get_issuing(

gta_instance_handle_t h_inst,

gta_access_token_t granting_token,

*gta_errinfo_t * p_errinfo)*

Get issuing token to enable the caller to issue basic access tokens.

This function is used to get an access token which can later be used to issue further access tokens.

This function can only be called once for every power on cycle of the device hosting GTA API. After this function has completed, any additional attempt to call the function fails. Whichever process has access to the returned access token can issue arbitrary basic access tokens (**gta_access_token_get_basic()**).

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
out	*granting_token*	Token granting the privilege to issue additional basic access tokens
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_FEATURE_NOT_SUPPORTED	This implementation of GTA API does not support basic access tokens.
GTA_ERROR_ACCESS	The current state of GTA API does not allow retrieval of an issuing token.

6.6.8.4.4 gta_access_token_get_physical_presence

bool gta_access_token_get_physical_presence (

gta_instance_handle_t h_inst,

gta_access_token_t physical_presence_token,

*gta_errinfo_t * p_errinfo)*

Get physical presence access token.

This function is used to get an access token which can later be used to prove physical presence.

This function can only be called once for every power-up cycle of the device hosting GTA API. After either this function or **gta_access_token_get_issuing()** has completed, any additional attempt to call this function fails.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
out	*physical_presence_ token*	Token granting the privilege to perform functionality that requires physical presence
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_FEATURE_NOT_SUPPORTED	This implementation of GTA API does not support basic access tokens.
GTA_ERROR_ACCESS	The current state of GTA API does not allow retrieval of an issuing token.

6.6.8.4.5 gta_access_token_revoke

bool gta_access_token_revoke (

gta_instance_handle_t h_inst,

gta_access_token_t access_token_tbr,

*gta_errinfo_t * p_errinfo)*

This function is used to invalidate an access token. Any process bearing the access token in question can invalidate the access token.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*access_token_tbr*	The access token to be invalidated
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_FEATURE_NOT_SUPPORTED	This implementation of GTA API does not support access tokens.
GTA_ERROR_ACCESS	The provided access token is invalid.

6.6.9 Device state management functions

6.6.9.1 Description

The functions in this group allow the management of GTA API device states.

6.6.9.2 Function documentation

6.6.9.2.1 gta_devicestate_attestate

bool gta_devicestate_attestate (

gta_context_handle_t h_context,

*gtaio_istream_t * nonce,*

*gtaio_ostream_t * attestation,*

*gta_errinfo_t * p_errinfo)*

Attest the genuineness of the top-most transition state.

Parameters:

in/out	**Name**	**Description**
in	*h_context*	Handle to the GTA API context; see **gta_context_open()** This context defines the personality and profile to be used to compute the attestation.
in	*nonce*	Nonce value to be used to prevent replay of an attestation
out	*attestation*	Attestation artifact
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.9.2.2 gta_devicestate_recede

bool gta_devicestate_recede (

gta_instance_handle_t h_inst,

gta_access_token_t access_token,

*gta_errinfo_t * p_errinfo)*

Recede into the previous transition device state (pop).

Parameters:

in/out	**Name**	**Description**
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*access_token*	Access token to authenticate the recede operation
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.9.2.3 gta_devicestate_transition

bool gta_devicestate_transition (

gta_instance_handle_t h_inst,

gta_access_policy_handle_t h_auth_recede,

size_t owner_lock_count,

*gta_errinfo_t * p_errinfo)*

Advance into a new transition device state (push).

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*h_auth_recede*	Access policy defining authentication required to run **gta_devicestate_recede()**; see **gta_access_policy_create()**
in	*owner_lock_count*	Counter to restrict the assignment of recede access policies allowing only personality derived authentication (that is, no authentication by physical access) for future device states; In case a transition state is created that cannot be removed using physical presence, the owner_lock_count shall be decremented as compared to previous device states. Once the counter is set to 0 for a device state, it is no longer allowed to prohibit physical presence authentication for following device states.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.10 Identifier and personality management

6.6.10.1 Description

This group of functions provides means to manage the personalities of the IIoT device which can be used by GTA API.

Personalities start from (insecure) device identifiers. Identifiers can be any string value practically usable to refer to a device. Examples are DNS names, GUIDs, URIs, X.500 names.

A personality provides a secure (usually enforced by cryptography) binding for an identifier. This binding can be used to prove that the device is authorized to act on behalf of the entity designated by the identifier. The personality also includes information on the parties that the entity is supposed to trust.

Personalities can be generated by a centralized service or supplied by a peer and deployed on the device (for example, X.509 PKI with centralized key generation). As an alternative to the centralized generation, personalities can be generated on the device and (optionally) shared with a peer or a central service (for example, X.509 PKI with decentralized key generation).

On an engineered device all installed identifiers and personalities can be known to the application upfront. Nevertheless, GTA API provides functionality to discover all objects known by GTA API using **gta_identifier_enumerate()**, **gta_personality_enumerate()**, and **gta_personality_enumerate_application()**. The example code snippet in Figure 18 illustrates this with **gta_personality_enumerate()**:

```
/* Enumerate all personalities for a given identifier_value
   within a GTA API instance h_inst. */

bool b_loop = true;
char namebuf[200];
myio_obufstream_t o_name = { 0 };
gta_enum_handle_t h_enum = GTA_HANDLE_ENUM_FIRST;

while (b_loop) {
    if (myio_open_obufstream(&o_name, namebuf, sizeof(namebuf),
            &errinfo)) {
        if (gta_personality_enumerate(h_inst, identifier_value,
                 &h_enum, GTA_PERSONALITY_ENUM_ALL,
                 (gtaio_ostream_t *)&o_name, &errinfo)) {
              /* ... do something with the
                       discovered personality ... */
         }
    }
    else {
        b_loop = false;
    }
    myio_close_obufstream(&o_name, &errinfo);
}
```

Figure 18 – Example gta_personality_enumerate()

6.6.10.2 Structure documentation

6.6.10.2.1 gta_ch_iec_30168_protection_properties_v0_t

struct gta_ch_iec_30168_protection_properties_v0_t {

bool integri;

bool intpers;

bool intmeta;

bool seccrea;

bool secread;

bool authuse;

bool authman;

bool authtru;

bool secextra;

bool secrepl;

}

Definition of protection properties according to the concept "ch.iec.30168.protection_properties.v0".

Member Data Documentation:

Type	Name	Description
bool	*integri*	Protection of integrity of secrets, trusted attributes, and security metadata against offline attacks
bool	*intpers*	Protection of integrity of a personality set, that is, the following mappings: • Between secret and trusted attributes • Between secret attributes, trusted attributes, and personalities • Between personalities and device states
bool	*intmeta*	Protection of security metadata
bool	*seccrea*	Securely create secret attributes on an SE
bool	*secread*	Ensure that secret attributes cannot be read from an SE when powered off
bool	*authuse*	Protection of use of secret and trusted attributes on an SE by using access tokens
bool	*authman*	Protection of management, for example, addition, of secret and trusted attributes by using device states
bool	*authtru*	Protection of validation based on trusted attributes
bool	*secextra*	Ensure that secrets cannot be extracted from an SE during operations that use them
bool	*recrepl*	Ensure that security-critical communication messages cannot be replayed

6.6.10.2.2 gta_protection_properties_t

```
struct gta_protection_properties_t {
    char * concept;
    union {
        struct gta_ch_iec_30168_protection_properties_v0_t
            ch_iec_30168_protection_properties_v0;
    };
};
```

Definition of protection properties applying to a personality stored on an SE.

Member Data Documentation:

Type	Name	Description
char *	*concept*	Semantic concept used to interpret the provided protection properties; GTA API includes the concept "ch.iec.30168.protection_properties.v0".
struct gta_ch_iec_30168_protection_properties_v0_t	*ch_iec_30168_protection_properties_v0*	Protection properties according to the concept "ch.iec.30168.protection_properties.v0"

6.6.10.3 Typedef documentation

6.6.10.3.1 gta_application_name_t

typedef char gta_application_name_t;*

Application name

Application defined classification criterion to group/identify personalities. Zero terminated string in UTF8 coding.

Apart from offering an enumeration operation that is based on this criterion, the application name can be freely chosen by the user and is handled transparently by GTA API.

6.6.10.3.2 gta_identifier_value_t

typedef char gta_identifier_value_t;*

Identifier value

Syntax and semantics of identifier values are defined by **gta_identifier_type_t**.

6.6.10.3.3 gta_identifier_type_t

typedef const char gta_identifier_type_t;*

Identifier type

Identifier type defines the syntax and semantics of an entity identifier that can be used by GTA API.

The following well-known identifier types are recognized:

Identifier	**Description**
ch.iec.30168.identifier.se_generic_hw_immutable	Immutable hardware identifier provided by the SE; This identifier type cannot be set by **gta_identifier_assign()** but is bound to the hardware during production. If this type of identifier is provided by the SE, it shall be returned from **gta_identifier_enumerate()**.
ch.iec.30168.identifier.uuid	String representation of an UUID, shall use the format as specified in RFC 4122, for example, f81d4fae-7dec-11d0-a765-00a0c91e6bf6
ch.iec.30168.identifier.dns_name	DNS name, shall use the format as specified in RFC 1035, for example, www.iec.ch
ch.iec.30168.identifier.x500_dn	X.500 distinguished name, shall use the format as specified in RFC 1779, for example, CN=L. Eagle, O="Sue, Grabbit and Runn", C=GB
ch.iec.30168.identifier.ipv4_addr	IPv4 address, for example, "130.23.14.5"
ch.iec.30168.identifier.ipv6_addr	IPv6 address, for example, 2001:0db8:0000:0000:0000:8a2e:0370:7334
ch.iec.30168.identifier.mac_addr	MAC address, shall use the format as specified in IEEE Std 802™-2014, for example, 01-23-45-67-89-AB

Identifier	Description
ch.iec.30168.identifier.uri	Uniform resource identifier, for example, https://example.com/path/resource.txt#fragment"
ch.iec.30168.identifier.generic	Deliberate string value - no semantics assumed.

6.6.10.3.4 gta_personality_attribute_name_t

typedef char gta_personality_attribute_name_t;*

Personality attribute name

Identification of an attribute belonging to a personality. Zero terminated string in UTF8 coding.

Attributes can be dynamically added and removed by the application.

6.6.10.3.5 gta_personality_attribute_type_t

typedef char gta_personality_attribute_type_t;*

Personality attribute type

Specification of an attribute type belonging to a personality. Zero terminated string in UTF8 coding.

The following well-known attribute types are defined; '*' is used to indicate that the given prefix should be used to define additional (mechanism) specific subtypes:

Attribute Type	Description
ch.iec.30168.identifier	The identifier value that is assigned to the personality at the time of its creation
ch.iec.30168.trustlist.certificate.self.x509	Public key certificate for the personalities end entity in X.509 ASN.1 DER coding
ch.iec.30168.trustlist.crl.x509v3	Certificate Revocation List (CRL) in X.509 ASN.1 DER coding.
ch.iec.30168.trustlist.certificate.trusted.x509v3	Trusted public key certificate in X.509 ASN.1 DER coding; Validation of a public key certificate chain can stop at this certificate. The public key certificate can be a self-signed root certificate or any other certificate that is directly trusted by the personality.
ch.iec.30168.trustlist.public_key.trusted.*	Any kind of (raw) trusted public key
ch.iec.30168.trustlist.psk_verifier.trusted.*	Any type of verifier that can be used to validate a pre-shared symmetric secret, for example, password authenticated key exchange (PAKE), challenge handshake authentication protocol (CHAP)
ch.iec.30168.trustlist.certificate.auxiliary.x509	Auxiliary public key certificate in X.509 ASN.1 distinguished encoding rules (DER) that are only trusted if additional evidence is provided; Auxiliary certificates can be used to construct a valid certificate chain ending in a trusted certificate.
ch.iec.30168.trustlist.certificate_list.rfc8446	Public key certificate chain, shall use the format as specified in RFC 8446, 4.4.2 (certificate_list)

6.6.10.3.6 gta_personality_enum_flags_t

```
typedef enum {
    GTA_PERSONALITY_ENUM_ALL = 0,
    GTA_PERSONALITY_ENUM_ACTIVE = 1,
    GTA_PERSONALITY_ENUM_INACTIVE = 2
} gta_personality_enum_flags_t;
```

Personality enumeration filtering, see **gta_personality_enumerate()**, **gta_personality_enumerate_application()**

Enumerator:

Enum	Description
GTA_PERSONALITY_ENUM_ALL	Enumerate all personalities
GTA_PERSONALITY_ENUM_ACTIVE	Enumerate only active personalities
GTA_PERSONALITY_ENUM_INACTIVE	Enumerate only deactivated personalities

6.6.10.3.7 gta_personality_fingerprint_t

typedef char gta_personality_fingerprint_t[64];

Personality fingerprint

64 bytes (512 bit) providing a unique fingerprint of the personality. The fingerprint can be used to create a robust binding to the personality within a cryptographic protocol.

6.6.10.3.8 gta_personality_name_t

typedef char gta_personality_name_t;*

Personality name

Identification for a personality. Zero terminated string in UTF8 coding.

The personality name shall be unique within an IIoT device.

Personalities are dynamically created and removed by the application according to specific profiles.

6.6.10.4 Function documentation

6.6.10.4.1 gta_identifier_assign

bool gta_identifier_assign (
    gta_instance_handle_t h_inst,
    const gta_identifier_type_t identifier_type,
    const gta_identifier_value_t identifier_value,
    *gta_errinfo_t * p_errinfo)*

Assign an identifier to the device.

There can be multiple identifiers. The value for the identifier shall be device unique regardless of the type.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle. **gta_instance_init()**
in	*identifier_type*	Type for the identifier being assigned
in	*identifier_value*	Value for the identifier being assigned
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.10.4.2 gta_identifier_enumerate

bool gta_identifier_enumerate (

gta_instance_handle_t h_inst,

*gta_enum_handle_t * ph_enum,*

*gtaio_ostream_t * identifier_type,*

*gtaio_ostream_t * identifier_value,*

*gta_errinfo_t * p_errinfo)*

Enumerate all identifiers managed by GTA API.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle. (**gta_instance_init()**)
in, out	*ph_enum*	Information to track the current enumeration. For the initial call starting the enumeration, the value that is referred to by ph_enum is GTA_HANDLE_ENUM_FIRST. Consecutive calls to continue the enumeration shall supply the value that was returned by the previous call. The handle is automatically closed after the last identifier has been returned and the next call to **gta_identifier_enumerate()** failed with **GTA_ERROR_ENUM_NO_MORE_ITEMS**. In case **gta_identifier_enumerate()** fails with any other error, the handle is also closed.
out	*identifier_type*	Type of the identifier enumerated
out	*identifier_value*	Value of the identifier enumerated
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_ENUM_NO_MORE_ITEMS	There are no more items after the last identifier has been enumerated.

6.6.10.4.3 gta_personality_add_attribute

bool gta_personality_add_attribute (

gta_context_handle_t h_ctx,

const gta_personality_attribute_type_t attrtype,

const gta_personality_attribute_name_t attrname,

*gtaio_istream_t * p_attrvalue,*

*gta_errinfo_t * p_errinfo)*

Assign an additional general attribute to an existing personality; see **gta_personality_create()**, **gta_personality_deploy()**

Potential types and semantics of general personality attributes are defined by the respective profile (for example, an X.509 end entity public key certificate).

General attributes can be deliberately added, modified, or removed. Security should not depend on the correctness of general attributes. Attributes critical for the security of the application should be stored as trusted attributes (**gta_personality_add_trusted_attribute()**).

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrtype*	Type of the general attribute
in	*attrname*	Unique name for the general attribute
in	*p_attrvalue*	Value for the general attribute
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_INVALID_ATTRIBUTE	The requested attribute is invalid or not present for the given personality.
GTA_ERROR_NAME_ALREADY_EXISTS	An attribute with the specified name already exists for this personality.

6.6.10.4.4 gta_personality_add_trusted_attribute

bool gta_personality_add_trusted_attribute (

gta_context_handle_t h_ctx,

const gta_personality_attribute_type_t attrtype,

const gta_personality_attribute_name_t attrname,

*gtaio_istream_t * p_attrvalue,*

*gta_errinfo_t * p_errinfo)*

Assign an additional trusted attribute to an existing personality; see **gta_personality_create()**, **gta_personality_deploy()**

Types and semantics of trusted personality attributes are defined by the respective profile (for example, an X.509 trusted root certificate).

Trusted attributes can only be added or updated after authentication as specified by the auth_admin policy that was specified in either **gta_personality_deploy()** or **gta_personality_create()**.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrtype*	Type of the trusted attribute
in	*attrname*	Unique name for the trusted attribute
in	*p_attrvalue*	Value for the trusted attribute
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_INVALID_ATTRIBUTE	The requested attribute is invalid or not present for the given personality.
GTA_ERROR_NAME_ALREADY_EXISTS	An attribute with the specified name already exists for this personality.
GTA_ERROR_ACCESS	The provided context does not hold the access token required to permit this operation.

6.6.10.4.5 gta_personality_attestate

bool gta_personality_attestate (

gta_context_handle_t h_ctx,

const gta_personality_name_t personality_name,

*gtaio_istream_t * nonce*

*gtaio_ostream_t * attestation_data,*

*gta_errinfo_t * p_errinfo)*

Attest properties of a personality.

This function is intended to testify that a given personality is compliant to specific security requirements (see **gta_protection_properties_t**). An example for such a requirement is that a private key providing the cryptographic binding for the personality has been generated in hardware on a specific SE (type) and was never extractable.

The resulting artifact can be supplied as attribute with a certificate signing request (**gta_personality_enroll()**).

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()** This context defines the personality and profile to be used to compute the attestation.
in	*personality_name*	The personality to which the properties to be testified belong
in	*nonce*	Nonce value to be used to prevent replay of an attestation
out	*attestation_data*	Data enabling a third party to prove the personality's properties which are subject to the attestation. The detailed format and semantics of this data are defined by the profile used
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.10.4.6 gta_personality_attributes_enumerate

bool gta_personality_attributes_enumerate (

gta_instance_handle_t h_inst,

const gta_personality_name_t personality_name,

*gta_enum_handle_t * ph_enum,*

*gtaio_ostream_t * attribute_type,*

*gtaio_ostream_t * attribute_name,*

*gta_errinfo_t * p_errinfo)*

Enumerate all attributes belonging to a personality.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*personality_name*	Name of the personality for which the available attributes are enumerated
in, out	*ph_enum*	Information to track the current enumeration; For the initial call starting the enumeration, the value that is referred to by ph_enum is GTA_HANDLE_ENUM_FIRST. Consecutive calls to continue the enumeration shall supply the value that was returned by the previous call. The handle is automatically closed after the last attribute has been returned and the next call to **gta_personality_attributes_enumerate()** failed with **GTA_ERROR_ENUM_NO_MORE_ITEMS**. In case **gta_personality_attributes_enumerate()** fails with any other error, the handle is also closed.
out	*attribute_type*	Type of the attribute enumerated
out	*attribute_name*	Name of the attribute enumerated
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	**Description**
GTA_ERROR_ENUM_NO_MORE_ITEMS	There are no more items after the last attribute has been enumerated.
GTA_ERROR_ITEM_NOT_FOUND	The supplied personality name is not valid.

6.6.10.4.7 gta_personality_create

bool gta_personality_create (

gta_instance_handle_t h_inst,

const gta_identifier_value_t identifier_value,

const gta_personality_name_t personality_name,

const gta_application_name_t application,

const gta_profile_name_t profile,

gta_access_policy_handle_t h_auth_use,

gta_access_policy_handle_t h_auth_admin,

struct gta_protection_properties_t requested_protection_properties,

*gta_errinfo_t * p_errinfo)*

Creates a personality on the device for a given identifier.

The credential to provide the cryptographic binding for the new personality is generated on the device, for example, generation of a new private key.

Parameters:

in/out	**Name**	**Description**
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*identifier_value*	Value of the identifier to which the new personality belongs
in	*personality_name*	Name to reference the new personality; The personality name uniquely identifies the personality within a given device.
in	*application*	Application defined filter criteria to discover the personality using **gta_personality_enumerate_application()**
in	*profile*	Profile indicating a type and format of the personality, for example, an ECDSA private key
in	*h_auth_use*	Access policy defining usage of the personality; see **gta_access_policy_create()**
in	*h_auth_admin*	Access policy defining administration of the personality; see **gta_access_policy_create()**
in	*requested_protection_properties*	Minimum requested protection properties to be provided for the new personality; If the requested protection properties cannot be met, the function shall fail. Successful execution of the function does not provide a proof that the security level is met. To prove the security level, an explicit attestation for the generated personality by the SE is required (see **gta_personality_attestate()**).
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_HANDLE_INVALID	The provided instance handle h_inst is invalid.
GTA_ERROR_NAME_ALREADY_EXISTS	A personality with the given name exists.
GTA_ERROR_PROFILE_UNSUPPORTED	The requested profile is either not supported by this function or there is no SE provider available to support this profile.

6.6.10.4.8 gta_personality_deploy

```
bool gta_personality_deploy (
    gta_instance_handle_t h_inst,
    const gta_identifier_value_t identifier_value,
    const gta_personality_name_t personality_name,
    const gta_application_name_t application,
    const gta_profile_name_t profile,
    gtaio_istream_t * personality_content,
    gta_access_policy_handle_t h_auth_use,
    gta_access_policy_handle_t h_auth_admin,
    struct gta_protection_properties_t requested_protection_properties,
    gta_errinfo_t * p_errinfo)
```

Installs a new personality with the given name into the device.

The information for the new personality is provisioned from an external source (containing, for example, a private key including its associated X.509 public key certificate or a symmetric pre-shared key).

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*identifier_value*	Value of the identifier to which the new personality belongs
in	*personality_name*	Name to reference the new personality. The personality name uniquely identifies the personality within a given device
in	*application*	Application defined filter criteria to discover the personality using **gta_personality_enumerate_application()**
in	*profile*	Profile for the personality; Used to specify type and format of the provided personality, for example, a PKCS#12 file including a private key and a well-defined set of certificates.
in	*personality_content*	Contents used to initialize the personality
in	*h_auth_use*	Access policy defining usage of the personality; see **gta_access_policy_create()**
in	*h_auth_admin*	Access policy defining administration of the personality; see **gta_access_policy_create()**

in/out	Name	Description
in	*requested_protection_properties*	Minimum requested protection properties to be provided for the new personality; If the requested protection properties cannot be met, the function shall fail. Successful execution of the function does not provide a proof that the security level is met. To prove the security level an explicit attestation for the generated personality by the SE is required (see **gta_personality_attestate()**).
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_HANDLE_INVALID	The provided instance handle h_inst is invalid.
GTA_ERROR_NAME_ALREADY_EXISTS	A personality with the given name exists.
GTA_ERROR_PROFILE_UNSUPPORTED	The requested profile is either not supported by this function or there is no SE provider available to support this profile.

6.6.10.4.9 gta_personality_enroll

bool gta_personality_enroll (

gta_context_handle_t h_ctx,

*gtaio_ostream_t * p_personality_enrollment_info,*

*gta_errinfo_t * p_errinfo)*

Creates an enrollment request for a personality.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()** The type, format, and content of the enrollment request (for example, CMP) and required attributes (for example, requested subject name, X.509 extensions) are defined by the profile that is specified opening the used context with **gta_context_open()**.
out	*p_personality_enrollment_info*	Information enabling the exchange of the created credential with a peer or third party; Depending on the given profile, for example, a certificate signing request, proof of possession, a key hash. Extra attributes required to create the enrollment can be provided either as persistent personality attributes (see **gta_personality_add_attribute()**) or temporary context attributes (see **gta_context_set_attribute()**) upfront. These extra attributes are described as part of the enrollment profile.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_HANDLE_INVALID	The provided context handle h_ctx is invalid.
GTA_ERROR_ATTRIBUTE_MISSING	A mandatory attribute that is required by the chosen profile to complete the enrollment request is missing.
GTA_ERROR_PROFILE_UNSUPPORTED	The requested profile is either not supported by this function or there is no SE provider available to support this profile.

6.6.10.4.10 gta_personality_enroll_auth

bool gta_personality_enroll_auth (

gta_context_handle_t h_ctx,

gta_context_handle_t h_auth_ctx,

*gtaio_ostream_t * p_personality_enrollment_info,*

*gta_errinfo_t * p_errinfo)*

Creates an authenticated enrollment request for personality.

In addition to **gta_personality_enroll()** this function supports the use of a dedicated authenticating context. Examples are countersigning certificate signing request using a personality already residing on the SE or binding the certificate signing request to a secure channel (for example, EST).

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*h_auth_ctx*	Context used to authenticate the created personality; The authenticating context is opened using an existing personality with **gta_context_open()**. In EST (enrollment over secure transport) scenarios, the authentication context must be successfully run through a handshake protocol using **gta_security_association_initialize()** and **gta_security_association_accept()**.
out	*p_personality_enrollment_info*	Information enabling the exchange of the created credential with a peer or third party; Depending on the given profile, for example, a certificate signing request, proof of possession, a key hash. Extra attributes required to create the enrollment can be provided either as persistent personality attributes (see **gta_personality_add_attribute()**) or temporary context attributes (see **gta_context_set_attribute()**) upfront. These extra attributes are described by the used enrollment profile.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.10.4.11 gta_personality_enumerate

bool gta_personality_enumerate (

gta_instance_handle_t h_inst,

const gta_identifier_value_t identifier_value,

*gta_enum_handle_t * ph_enum,*

gta_personality_enum_flags_t flags,

*gtaio_ostream_t * personality_name,*

*gta_errinfo_t * p_errinfo)*

Enumerate all personalities that are known to GTA API by their identifier.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*identifier_value*	Identifier for which the available personalities are enumerated
in, out	*ph_enum*	Information to track the current enumeration; For the initial call starting the enumeration, the value that is referred to by ph_enum is GTA_HANDLE_ENUM_FIRST. Consecutive calls to continue the enumeration shall supply the value that was returned by the previous call. The handle is automatically closed after the last personality has been returned and the next call to **gta_personality_enumerate()** failed with **GTA_ERROR_ENUM_NO_MORE_ITEMS**. In case **gta_personality_enumerate()** fails with any other error, the handle is also closed.
in	*flags*	Allows to select between active and deactivated personalities
out	*personality_name*	Name of the personality enumerated
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_ENUM_NO_MORE_ITEMS	There are no more items after the last personality has been enumerated.
GTA_ERROR_ITEM_NOT_FOUND	The supplied identifier is not valid.

6.6.10.4.12 gta_personality_enumerate_application

bool gta_personality_enumerate_application (

gta_instance_handle_t h_inst,

const gta_application_name_t application_name,

*gta_enum_handle_t * ph_enum,*

gta_personality_enum_flags_t flags,

*gtaio_ostream_t * personality_name,*

*gta_errinfo_t * p_errinfo)*

Enumerate all personalities that are known to GTA API by their application name.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*application_name*	Application defined classification criterion to group personalities for which the available personalities are enumerated
in, out	*ph_enum*	Information to track the current enumeration; For the initial call starting the enumeration, the value that is referred to by ph_enum is GTA_HANDLE_ENUM_FIRST. Consecutive calls to continue the enumeration shall supply the value that was returned by the previous call. The handle is automatically closed after the last personality has been returned and the next call to **gta_personality_enumerate()** failed with **GTA_ERROR_ENUM_NO_MORE_ITEMS**. In case **gta_personality_enumerate()** fails with any other error, the handle is also closed.
in	*flags*	Allows selection between active and deactivated personalities
out	*personality_name*	Name of the personality enumerated
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_ENUM_NO_MORE_ITEMS	There are no more items after the last personality has been enumerated.
GTA_ERROR_ITEM_NOT_FOUND	The supplied application name is not present.

6.6.10.4.13 gta_personality_get_attribute

bool gta_personality_get_attribute (

gta_context_handle_t h_ctx,

const gta_personality_attribute_name_t attrname,

*gtaio_ostream_t * p_attrvalue,*

*gta_errinfo_t * p_errinfo)*

Get attribute of a personality.

Types and semantics of attributes are defined by the respective profiles.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrname*	Name identifying the attribute to be queried
out	*p_attrvalue*	Stream receiving the attribute value
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_INVALID_ATTRIBUTE	The requested attribute is invalid or not present for the given personality.

The following attributes are mandatory for every personality:

Attribute Name	Description
ch.iec.30168.identifier_value	Value that is passed as identifier_value to **gta_personality_create()** or **gta_personality_deploy()**; Zero terminated string
ch.iec.30168.fingerprint	Fingerprint of the personality; 64 octets

6.6.10.4.14 gta_personality_remove_attribute

bool gta_personality_remove_attribute (

gta_context_handle_t h_ctx,

const gta_personality_attribute_name_t attrname,

*gta_errinfo_t * p_errinfo)*

Remove an attribute from a personality.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrname*	Name identifying the attribute to be removed
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.10.4.15 gta_personality_remove

bool gta_personality_remove (

gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Remove a personality from the device.

After calling gta_personality_remove(), all subsequent calls to GTA API using a context referring to the respective personality shall fail (apart from gta_context_close()).

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()** The profile used to remove a personality is typically the same profile that has been used to create the personality.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.10.4.16 gta_personality_deactivate

bool gta_personality_deactivate (

gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Deactivate a personality.

This function temporarily deactivates a personality. The personality can be activated again using **gta_personality_activate()**.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.10.4.17 gta_personality_activate

bool gta_personality_activate (

gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Activate a previously deactivated personality.

This function activates a personality which has been deactivated by **gta_personality_deactivate()** before.

The authentication policy that is given as auth_admin in **gta_personality_create()** or **gta_personality_deploy()** must be fulfilled as a prerequisite for successful completion of this function.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_ACCESS	The provided context does not hold the access token that is required to permit this operation.

6.6.10.4.18 gta_personality_deactivate_attribute

bool gta_personality_deactivate_attribute (

gta_context_handle_t h_ctx,

const gta_personality_attribute_name_t attrname,

*gta_errinfo_t * p_errinfo)*

Deactivate a personality attribute.

This function can be used to temporarily deactivate a personality attribute. The attribute can be activated again using **gta_personality_activate_attribute()**.

The authentication policy that is given as auth_admin in **gta_personality_create()** or **gta_personality_deploy()** must be fulfilled as a prerequisite for successful completion of this function for a trusted attribute.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrname*	Name identifying the attribute to be removed
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_INVALID_ATTRIBUTE	The requested attribute is invalid or not present for the given personality.
GTA_ERROR_ACCESS	The provided context does not hold the access token that is required to permit this operation.

6.6.10.4.19 gta_personality_activate_attribute

bool gta_personality_activate_attribute (

gta_context_handle_t h_ctx,

const gta_personality_attribute_name_t attrname,

*gta_errinfo_t * p_errinfo)*

Activate a previously deactivated personality attribute.

This function allows activation of a personality after it has been deactivated using **gta_personality_deactivate_attribute()**.

The authentication policy that is given as auth_admin in **gta_personality_create()** or **gta_personality_deploy()** must be fulfilled as a prerequisite for successful completion of this function for a trusted attribute.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*attrname*	Name identifying the attribute to be removed
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_INVALID_ATTRIBUTE	The requested attribute is invalid or not present for the given personality.
GTA_ERROR_ACCESS	The provided context does not hold the access token that is required to permit this operation.

6.6.11 Access policy management functions

6.6.11.1 Description

These functions are used for the definition of access policies that are assigned to personalities with **gta_personality_create()**, **gta_personality_deploy()**, and device states with **gta_devicestate_transition()**.

Access policies are set up by the application using GTA API. Simple access policies (specifying a single access condition) can be created using the function:

- **gta_access_policy_simple()**

More complex access policies can be constructed using the functions:

- **gta_access_policy_create()**
- **gta_access_policy_add_physical_presence_access_token_descriptor()**
- **gta_access_policy_add_basic_access_token_descriptor()**
- **gta_access_policy_add_pers_derived_access_token_descriptor()**
- **gta_access_policy_destroy()**

GTA API forwards the access policies to the SE provider. GTA API SE providers can parse these provided access policies using the functions:

- **gta_access_policy_enumerate()**,
- **gta_access_policy_get_access_descriptor_type()**, and
- **gta_access_policy_get_access_descriptor_attribute()**.

Based on the information that is obtained in the access policy the SE provider enforces the access control. The enforcement relies on the native access control mechanisms of the SE. The example code snippet in Figure 19 illustrates this process for a usage access policy passed to **gta_personality_deploy()**.

```
bool myprovider_gta_personality_deploy(
         …,
         gta_access_policy_handle_t h_auth_use,
         gta_access_policy_handle_t h_auth_admin,
         …,
         gta_errinfo_t * p_errinfo
)
{
    ...

    gta_enum_handle_t h_enum = GTA_HANDLE_ENUM_FIRST;
    gta_access_descriptor_handle_t
        h_access_descriptor = GTA_HANDLE_INVALID;
    char * p_attr = NULL;
    size_t len = 0;

    while gta_access_policy_enumerate(
        h_auth_use, &h_enum, &h_access_descriptor,
        p_errinfo))
    {
        gta_access_descriptor_type_t
            access_descriptor_type;

        if (gta_access_policy_get_access_descriptor_type(h_auth_use,
                h_access_descriptor, &access_descriptor_type,
                p_errinfo))
        {
            switch (access_descriptor_type) {
            case GTA_ACCESS_DESCRIPTOR_TYPE_INITIAL:
                /* setup initial access protection for native SE object */
                ...
                break;
            case GTA_ACCESS_DESCRIPTOR_TYPE_BASIC_TOKEN:
                /* setup basic access protection for native SE object */
                ...
                break;
            case GTA_ACCESS_DESCRIPTOR_TYPE_PERS_DERIVED_TOKEN:
                /* setup personality derived access protection
                   according to its attributes */
                if (gta_access_policy_get_access_descriptor_attribute(
                        h_auth_use,
                        h_access_descriptor,
                        GTA_ACCESS_TOKEN_DESCRIPTOR_ATTR_PROFILE_NAME,
                        &p_attr, &len,
                        p_errinfo
                   )) {
                    ...
                }
                ...
                break;
            case ...
            default:
                *p_errinfo = GTA_ERROR_INVALID_PARAMETER;
                goto err;
                break;
            }
        }
    }

    ...
}
```

Figure 19 – Example access policy handling by SE provider

6.6.11.2 Enumeration type documentation

6.6.11.2.1 gta_access_policy_handle_t

typedef gta_handle_t gta_access_policy_handle_t

Access policy handle

6.6.11.2.2 gta_access_descriptor_attribute_type_t

```
typedef enum {
    GTA_ACCESS_DESCRIPTOR_ATTR_PROFILE_NAME = 1,
    GTA_ACCESS_DESCRIPTOR_ATTR_PERS_FINGERPRINT = 2
} gta_access_descriptor_attribute_type_t;
```

Attribute types used within access descriptors.

Enumerator:

Enum	Description
GTA_ACCESS_DESCRIPTOR_ATTR_PROFILE_NAME	Attribute value is a reference to a profile by its profile name (see **gta_profile_name_t**).
GTA_ACCESS_DESCRIPTOR_ATTR_PERS_FINGERPRINT	Attribute value is a reference to a personality by its fingerprint (see **gta_personality_fingerprint_t**).

6.6.11.2.3 gta_access_descriptor_type_t

```
typedef enum {
    GTA_ACCESS_DESCRIPTOR_TYPE_INITIAL = 0,
    GTA_ACCESS_DESCRIPTOR_TYPE_BASIC_TOKEN = 1,
    GTA_ACCESS_DESCRIPTOR_TYPE_PERS_DERIVED_TOKEN = 2,
    GTA_ACCESS_DESCRIPTOR_TYPE_PHYSICAL_PRESENCE_TOKEN = 3
} gta_access_descriptor_type_t;
```

Definitions for predefined access policies.

Enumerator:

Enum	Description
GTA_ACCESS_DESCRIPTOR_TYPE_INITIAL	Initial access applies for this access policy.
GTA_ACCESS_DESCRIPTOR_TYPE_BASIC_TOKEN	A basic access token is required to meet this access policy.
GTA_ACCESS_DESCRIPTOR_TYPE_PERS_DERIVED_TOKEN	A personality derived access token is required to meet this access policy.
GTA_ACCESS_DESCRIPTOR_TYPE_PHYSICAL_PRESENCE_TOKEN	A physical presence access token is required to meet this access policy.

6.6.11.2.4 gta_access_token_usage_t

typedef enum {

GTA_ACCESS_TOKEN_USAGE_USE = 0,

GTA_ACCESS_TOKEN_USAGE_ADMIN = 1,

GTA_ACCESS_TOKEN_USAGE_RECEDE = 2

} gta_access_token_usage_t;

Definitions for access token usages.

Enumerator:

Enum	Description
GTA_ACCESS_TOKEN_USAGE_USE	Access token applies to the usage of an object.
GTA_ACCESS_TOKEN_USAGE_ADMIN	Access token applies to the administration of an object.
GTA_ACCESS_TOKEN_USAGE_RECEDE	Access token applies to a recede operation on a device state.

6.6.11.3 Function documentation

6.6.11.3.1 gta_access_policy_add_basic_access_token_descriptor

bool gta_access_policy_add_basic_access_token_descriptor (

gta_access_policy_handle_t h_access_policy,

*gta_errinfo_t * p_errinfo)*

Add a descriptor for a basic access token to the policy (only applicable for access policies that are to be assigned to personalities).

Parameters:

in/out	Name	Description
in	*h_access_policy*	Handle to the access policy object.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.11.3.2 gta_access_policy_add_pers_derived_access_token_descriptor

bool gta_access_policy_add_pers_derived_access_token_descriptor (

gta_access_policy_handle_t h_access_policy,

const gta_personality_fingerprint_t personality_fingerprint,

const gta_profile_name_t verification_profile_name,

*gta_errinfo_t * p_errinfo)*

Add a descriptor for a personality derived access token to the policy.

Parameters:

in/out	Name	Description
in	*h_access_policy*	Handle to the access policy object
in	*personality_fingerprint*	Fingerprint of the personality that is to be used to derive an access token; The fingerprint provides a cryptographic binding so that only this unique personality can be used to derive a valid access token. The fingerprint can be set to the personality name of the personality to which the access policy is attached to create a self-reference. The personality name is either padded with '\0' or truncated to fit into gta_personality_fingerprint_t.
in	*verification_profile_name*	Name of the profile that is to be used to derive the access token; This profile name is bound to any resulting access token to decide whether the correct profile has been used to derive the token.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.11.3.3 gta_access_policy_add_physical_presence_access_token_descriptor

bool gta_access_policy_add_physical_presence_access_token_descriptor (

gta_access_policy_handle_t h_access_policy,

*gta_errinfo_t * p_errinfo)*

Add physical presence to an access policy (only applicable for access policies that are to be assigned to device states).

Parameters:

in/out	Name	Description
in	*h_access_policy*	Handle to the access policy object
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.11.3.4 gta_access_policy_create

gta_access_policy_handle_t gta_access_policy_create (

gta_instance_handle_t h_inst,

*gta_errinfo_t * p_errinfo)*

Create an access policy.

The conditions that shall be met by an access token in order to fulfil an access policy are specified using **gta_access_policy_add_basic_access_token_descriptor()** and **gta_access_policy_add_pers_derived_access_token_descriptor()**.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Handle to the created access policy object or GTA_HANDLE_INVALID if there is an error. The returned handle can be released using **gta_access_policy_destroy()**.

6.6.11.3.5 gta_access_policy_destroy

bool gta_access_policy_destroy (

gta_access_policy_handle_t h_access_policy,

*gta_errinfo_t * p_errinfo)*

Destroy an access policy.

This function is used to release all resources that are linked to an access policy object created using **gta_access_policy_create()**. An access policy object shall not be destroyed if any GTA API function is still using the access policy object.

It is not required to release access policies created with **gta_access_policy_simple()**.

Parameters:

in/out	Name	Description
in	*h_access_policy*	Handle to access policy object to be destroyed
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.11.3.6 gta_access_policy_get_access_descriptor_attribute

bool gta_access_policy_get_access_descriptor_attribute (

gta_access_descriptor_handle_t h_access_descriptor,

gta_access_descriptor_attribute_type_t attr_type,

*const char ** pp_attr,*

*size_t * p_attr_len,*

*gta_errinfo_t * p_errinfo)*

Get the specified attribute of an access token descriptor.

The available attribute types depend on the access type **gta_access_policy_get_access_descriptor_type()**.

Parameters:

in/out	Name	Description
in	*h_access_descriptor*	Handle to the access descriptor
in	*attr_type*	Type identifying the attribute which value is queried.
in, out	*pp_attr*	Pointer to the memory location that receives a pointer to the attribute value; It is safe to use the returned pointer as long as the underlying access_policy object is not destroyed using **gta_access_policy_destroy()**.
in, out	*p_attr_len*	Pointer to the memory location that receives the length of the attribute in bytes; In case the attribute is a string value, the terminating '\0' is not included.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.11.3.7 gta_access_policy_get_access_descriptor_type

bool gta_access_policy_get_access_descriptor_type (
gta_access_policy_handle_t h_access_policy,
gta_access_descriptor_handle_t h_access_descriptor,
*gta_access_descriptor_type_t * p_access_descriptor_type,*
*gta_errinfo_t * p_errinfo)*

Get type of an access descriptor.

Parameters:

in/out	Name	Description
in	*h_access_policy*	Handle to the access policy object
in	*h_access_descriptor*	Handle to the access descriptor of which the type is queried
in, out	*p_access_descriptor_type*	Location receiving the access descriptor which is one of **gta_access_descriptor_type_t**.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.11.3.8 gta_access_policy_simple

gta_access_policy_handle_t gta_access_policy_simple
(gta_instance_handle_t h_inst,
gta_access_descriptor_type_t access_descriptor_type,
*gta_errinfo_t * p_errinfo)*

Get handle for a simple (static) access policy.

This function provides a shortcut to setup a simple access policy consisting of a single access token. The single access token can be of type GTA_ACCESS_ DESCRIPTOR_TYPE_INITIAL, GTA_ACCESS_DESCRIPTOR_TYPE_BASIC_TOKEN, or GTA_ACCESS_DESCRIPTOR_TYPE_PHYSICAL_PRESENCE_TOKEN (see **gta_access_descriptor_type_t**)

This function provides a straightforward way to set up commonly used simple access policies as an alternative to using **gta_access_policy_create()**, **gta_access_policy_add_...()**, and **gta_access_policy_destroy()**.

It is not possible to add additional access token descriptors to a static access policy using **gta_access_policy_add_...()** functions.

Any attempt to release the returned handle using **gta_access_policy_destroy()** should result in an GTA_ERROR_HANDLE_INVALID error.

Parameters:

in/out	Name	Description
in	*h_inst*	GTA API instance handle; see **gta_instance_init()**
in	*access_ descriptor_type*	Type of single access descriptor that is used to setup the simple access policy; GTA_ACCESS_DESCRIPTOR_TYPE_INITAL, GTA_ACCESS_ DESCRIPTOR_TYPE_BASIC_TOKEN, or GTA_ACCESS_ DESCRIPTOR_TYPE_PHYSICAL_PRESENCE_TOKEN; see **gta_access_descriptor_type_t**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Handle to an access policy or GTA_HANDLE_INVALID on failure.

6.6.11.3.9 gta_access_policy_enumerate

bool gta_access_policy_enumerate (

gta_access_policy_handle_t h_access_policy,

*gta_enum_handle_t * ph_enum,*

*gta_access_descriptor_handle_t * ph_access_descriptor,*

*gta_errinfo_t * p_errinfo)*

Enumerate all access descriptors defining the access policy.

Parameters:

in/out	Name	Description
in	*h_access_policy*	Handle to the access policy object to be enumerated
in, out	*ph_enum*	Information to track the current enumeration; For the initial call starting the enumeration, the value that is referred to by ph_enum is GTA_HANDLE_ENUM_FIRST. Consecutive calls to continue the enumeration shall supply the value that was returned by the previous call. The handle is automatically closed after the last personality has been returned and the next call to **gta_access_policy_enumerate()** failed with **GTA_ERROR_ENUM_NO_MORE_ITEMS**. In case **gta_access_policy_enumerate()** fails with any other error, the handle is also closed.
in, out	*ph_access_descriptor*	Pointer to a memory location that receives the next access descriptor handle; The handle stays valid until the respective access_policy object is destroyed.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_ENUM_NO_MORE_ITEMS	There are no more items after the last identifier has been enumerated.

6.6.12 Data protection functions

6.6.12.1 Description

The following group of functions addresses the protection of persistent/in-situ data. Protection is either archived by putting the data into a secure container/envelope (**gta_seal_data()**, **gta_unseal_data()**) or by computing a detached seal (**gta_authenticate_data_detached()**, **gta_verify_data_detached()**). If a secure container/envelope is used, **gta_unseal_data()** recovers the original data from the protected data BLOB. If a detached seal is used, the original data must be supplied to the verification function (**gta_verify_data_detached()**) as it is not carried with the authentication information.

6.6.12.2 Function documentation

6.6.12.2.1 gta_authenticate_data_detached

bool gta_authenticate_data_detached (

gta_context_handle_t h_ctx,

*gtaio_istream_t * data,*

*gtaio_ostream_t * seal,*

*gta_errinfo_t * p_errinfo)*

Calculates a cryptographic seal for the provided data according to the profile and personality that is given by the specified context.

Unlike **gta_seal_data()/gta_unseal_data()** the data cannot be recovered from the output returned.

NOTE The function name and signature suggest that this functionality does only support integrity protection/authentication. Detached protection information does not seem sensible for encryption/decryption.

Depending on the definition in the profile, an access token granting usage authentication is required.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*data*	Stream providing the data to be protected
out	*seal*	Stream receiving the protection seal
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.12.2.2 gta_seal_data

bool gta_seal_data (

gta_context_handle_t h_ctx,

*gtaio_istream_t * data,*

*gtaio_ostream_t * protected_data,*

*gta_errinfo_t * p_errinfo)*

Protect a piece of data according to the profile and personality that is given by the specified context.

The type of protection, for example, authentication, integrity protection, encryption is defined by the chosen profile.

Depending on the definition in the profile, an access token granting usage authentication is required.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*data*	Stream providing the data to be protected
out	*protected_data*	Stream receiving the protected data
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.12.2.3 gta_unseal_data

bool gta_unseal_data (

gta_context_handle_t h_ctx,

*gtaio_istream_t * protected_data,*

*gtaio_ostream_t * data,*

*gta_errinfo_t * p_errinfo)*

Recover a piece of data according to the profile and personality that is given by the specified context.

Depending on the definition in the profile, an access token granting usage authentication is required.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*protected_data*	Stream providing the protected data
out	*data*	Stream receiving the result of the unseal operation
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.12.2.4 gta_verify_data_detached

bool gta_verify_data_detached (

gta_context_handle_t h_ctx,

*gtaio_istream_t * data,*

*gtaio_istream_t * seal,*

*gta_errinfo_t * p_errinfo)*

Verify a cryptographic seal for the provided data according to the profile and personality that is given by the specified context.

Unlike **gta_seal_data()**/**gta_unseal_data()** the data cannot be recovered from the output returned.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*data*	Stream providing the data to be verified
in	*seal*	Authentication seal to be verified
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.12.2.5 gta_verify

bool gta_verify (

gta_context_handle_t h_ctx,

*gtaio_istream_t * claim,*

*gta_errinfo_t * p_errinfo)*

Verify a claim according to the profile and personality that is given by the specified context.

This function is intended to verify claims against information belonging to a personality. Examples for claims are passcodes, responses to challenges, public keys, or public key certificate chains. The respective verification information belonging to a personality for the given examples are passcode verifiers, challenge verifiers, trusted keys, or trusted certificates.

ch.iec.30168.basic.passcode (Clause B.1) is an example for passcode-based authentication. Profiles depending on other claims can be defined in the future.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*claim*	Stream providing the claim to be verified
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.13 Channel protection functions

6.6.13.1 Description

The following group of functions refers to the creation and use of a protected communication channel that can be used to exchange data.

Data exchange, for example, using a communication network, is dealt with outside GTA API. The GTA API mechanism for passing input and output parameters are streams (see 6.6.5). Use of streams within the channel protection context allows to minimize unnecessary I/O or buffer operations, for example, by directly passing a network stream to GTA API.

Figure 20 illustrates the establishment of use of a security association using GTA API functions.

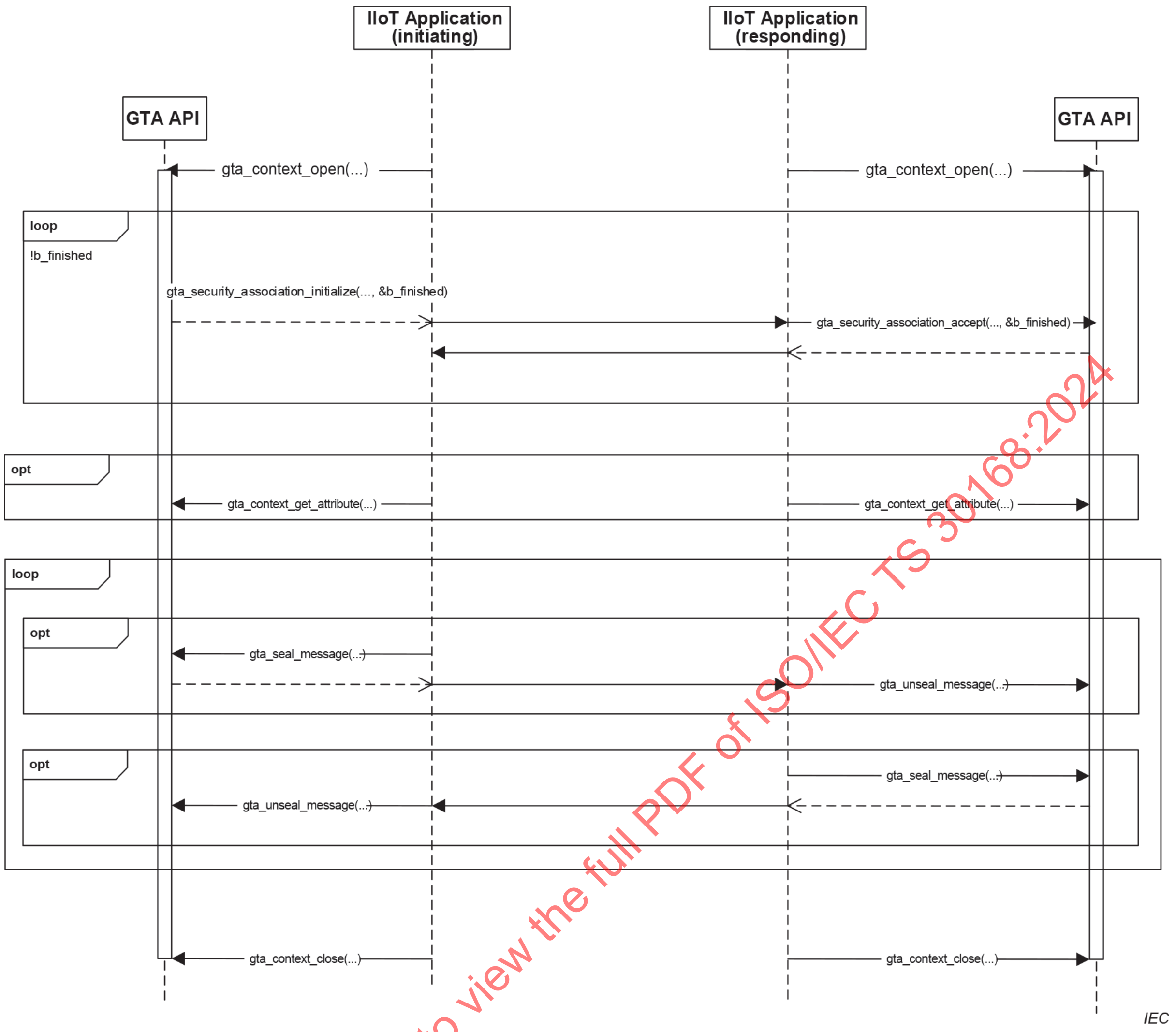

IIoT Application (initiating)
IIoT Application (responding)
GTA API
GTA API
gta_context_open(...)
gta_context_open(...)
loop
!b_finished
gta_security_association_initialize(..., &b_finished)
gta_security_association_accept(..., &b_finished)
opt
gta_context_get_attribute(...)
gta_context_get_attribute(...)
loop
opt
gta_seal_message(...)
gta_unseal_message(...)
opt
gta_seal_message(...)
gta_unseal_message(...)
gta_context_close(...)
gta_context_close(...)
IEC

Figure 20 – Channel protection functions

Depending on the profile, it is required to call **gta_security_association_initialize()** and **gta_security_association_accept()** multiple times to complete security protocols requiring multiple roundtrips.

After the security association has been established successfully, attributes of the security association can be queried using **gta_context_get_attribute()**. Which attributes are defined depends on the respective GTA API profile. Examples are:

- Expiration time of the association
- Channel binding information

The functions **gta_seal_message()** and **gta_unseal_message()** can be used to exchange messages protected by the security association. Use of these functions eliminates the need to expose secret or sensitive information belonging to the security association outside GTA API.

6.6.13.2 Function documentation

6.6.13.2.1 gta_seal_message

bool gta_seal_message (

gta_context_handle_t h_ctx,

*gtaio_istream_t * msg,*

*gtaio_ostream_t * sealed_msg,*

*gta_errinfo_t * p_errinfo)*

Seal a message to be transferred using the security association of the given context.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*msg*	Stream providing the message data to be protected
out	*sealed_msg*	Stream receiving the protected data
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.13.2.2 gta_security_association_accept

bool gta_security_association_accept (

gta_context_handle_t h_ctx,

*gtaio_istream_t * in,*

*gtaio_ostream_t * out,*

*bool * pb_finished,*

*gta_errinfo_t * p_errinfo)*

Initialize a cryptographic association that is shared between two or more entities.

This function is used by the entity responding to a secure connection request (see Figure 20).

There can be only one concurrent security association per context.

Depending on the definition in the profile, an access token granting usage authentication is required.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*in*	Input data for the current protocol step
out	*out*	Output data of the current protocol step
out	*pb_finished*	Indicates whether additional calls are required
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_CONTEXT_BUSY	There is already a security association using this context. Either close the existing security association **gta_security_association_destroy()** or open a new context.

6.6.13.2.3 gta_security_association_destroy

bool gta_security_association_destroy (

gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Destroy an existing security association.

Destroy a security association that was created by either **gta_security_association_initialize()** or **gta_security_association_accept()** and all its related resources.

After the security association has been destroyed, a new security association may be established using the same context.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.13.2.4 gta_security_association_initialize

bool gta_security_association_initialize (

gta_context_handle_t h_ctx,

*gtaio_istream_t * in,*

*gtaio_ostream_t * out,*

*bool * pb_finished,*

*gta_errinfo_t * p_errinfo)*

Initialize a cryptographic association that is shared between two or more entities.

This function is used by the entity initiating a security protocol by sending a secure connection request (see Figure 20).

There can be only one concurrent security association per context.

Depending on the definition in the profile, an access token granting usage authentication is required.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*in*	Input data for the current protocol step
out	*out*	Output data of the current protocol step
out	*pb_finished*	Indicates whether additional calls are required
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

Error Information	Description
GTA_ERROR_CONTEXT_BUSY	There is already a security association using this context. Either close the existing security association **gta_security_association_destroy()** or open a new context.

6.6.13.2.5 gta_unseal_message

bool gta_unseal_message (

*gta_context_handle_t * h_ctx,*

*gtaio_istream_t * sealed_msg,*

*gtaio_ostream_t * msg,*

*gta_errinfo_t * p_errinfo)*

Unseal a message received using the given security association of the given context.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
in	*sealed_msg*	Stream providing the sealed message data to be unprotected
out	*msg*	Stream receiving the resulting data
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.14 Supplementary security functions

6.6.14.1 Description

The following group of functions provides functionality which is typically supported by SEs but does not depend on a distinct context or personality. One such function is the retrieval of random numbers which can be used as seeds, nonces, or challenges within security protocols.

6.6.14.2 Function documentation

6.6.14.2.1 gta_get_random_bytes

bool gta_get_random_bytes (

size_t num_bytes,

*gtaio_ostream_t * rnd_stream,*

*gta_errinfo_t * p_errinfo)*

Get some random bytes.

This function writes num_bytes of random into the provided stream.

Parameters:

in/out	**Name**	**Description**
in	*num_bytes*	Number of random bytes being requested.
out	*rnd_stream*	Stream that receives the random bytes.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.14.2.2 gta_attestate

bool gta_attestate (

gta_context_handle_t h_ctx,

*gtaio_istream_t * nonce*

*gtaio_ostream_t * attestation_data,*

*gta_errinfo_t * p_errinfo)*

Attest properties of a GTA API implementation or device.

This function is intended to testify that a given GTA API implementation or device is compliant to specific security requirements.

The properties to be testified can be but are not limited to those defined by **gta_protection_properties_t**. Another example is any kind of measurement of the device GTA API is running on.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()** This context defines the personality and profile to be used to compute the attestation.
in	*nonce*	Nonce value to be used to prevent replay of an attestation
out	*attestation_data*	Data enabling a third party to prove the personalities properties which are subject to the attestation; The detailed format and semantics of this data are defined by the profile used.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.15 Trusted execution environment

6.6.15.1 Description

The following group of functions that is used to manage and run functions inside a trusted execution environment that is provided on the SE.

6.6.15.2 Functions

- bool **gta_trustex_function_install** (const char *function_name, gta_profile_name_t profile_name, gtaio_istream_t function, gta_errinfo_t * p_errinfo)
- bool **gta_trustex_function_uninstall** (const char *function_name, gta_errinfo_t * p_errinfo)
- bool **gta_trustex_function_execute** (const char *function_name, gta_handle_t function_handle, gtaio_istream_t input, gtaio_ostream_t output, gta_errinfo_t * p_errinfo)
- bool **gta_trustex_function_terminate** (gta_handle_t function handle, gta_errinfo_t * p_errinfo)

6.6.16 Secure element provider implementation support

6.6.16.1 Description

The following functions are for the exclusive use for the implementation of SE providers. The functions are primarily used by providers to manage information managed by the GTA API on their behalf.

6.6.16.2 Function documentation

6.6.16.2.1 gta_provider_context_close

bool gta_provider_context_close (

gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Close an existing context.

This provider-specific function is called by the GTA API framework at the beginning of **gta_context_close()**. **gta_provider_context_close()** allows the SE provider to finalize any provider-specific resources associated with the context. See also **gta_context_open()**, **gta_provider_context_open()**.

Dynamic memory that was allocated by **gta_secmem_calloc()** using the context to be closed will automatically be released by the GTA API framework during the execution of **gta_context_close()**. The memory is released after the provider specific function **gta_provider_context_close()** returned.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.16.2.2 gta_provider_context_open

bool gta_provider_context_open (

gta_context_handle_t h_ctx,

const gta_personality_name_t personality,

const gta_profile_name_t profile,

*void ** pp_params, gta_errinfo_t * p_errinfo)*

Open a new context.

Contexts are used to organize operations relating to a specific personality, for example, a key agreement, data protection/unprotection.

gta_context_open() will call this provider-specific function after preparing a new context that can be used by the provider.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the new context; The context is created and initialized by the GTA API framework and can directly be used by the provider.
in	*personality*	Personality to use for the context
in	*profile*	Profile to use for the context
in, out	*pp_params*	Pointer to accept a reference to context local parameters; Functions that are implemented by the provider can later access this reference using **gta_context_get_ params()**.
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

True on success, false on failure

6.6.16.2.3 gta_provider_get_params

void gta_provider_get_params (*

gta_instance_handle_t h_inst,

*gta_errinfo_t * p_errinfo)*

Return provider instance global parameters.

This function can be used by the implementation of a provider to access the information that is stored within pp_params returned by the providers initialization function **gta_provider_init_t**.

gta_context_get_provider_params() returns the same information using a context handle instead of an instance handle.

Parameters:

in/out	Name	Description
in	*h_inst*	Handle to the GTA API instance; see **gta_instance_init()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Pointer to the instance parameters or NULL on failure.

6.6.16.2.4 gta_context_get_provider_params

void gta_context_get_provider_params (*

gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Return provider instance global parameters.

This function can be used by the implementation of a provider to access the information stored within pp_params returned by the providers initialization function **gta_provider_init_t**.

gta_provider_get_params() returns the same information using an instance handle instead of a context handle.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Pointer to the instance parameters or NULL on failure.

6.6.16.2.5 gta_context_get_params

void gta_context_get_params*

(gta_context_handle_t h_ctx,

*gta_errinfo_t * p_errinfo)*

Return context-specific parameters.

This function can be used by the provider implementation of a provider to access the information that is stored within pp_params returned when opening a context **gta_provider_context_open()**.

Parameters:

in/out	Name	Description
in	*h_ctx*	Handle to the GTA API context; see **gta_context_open()**
out	*p_errinfo*	Detailed error information if there is a failure

Returns:

Pointer to the context parameters or NULL on failure.

Annex A
(normative)

GTA API C header files

A.1 Dependencies

Figure A.1 shows an overview of all GTA API header files (`gta_...`) and their dependencies.

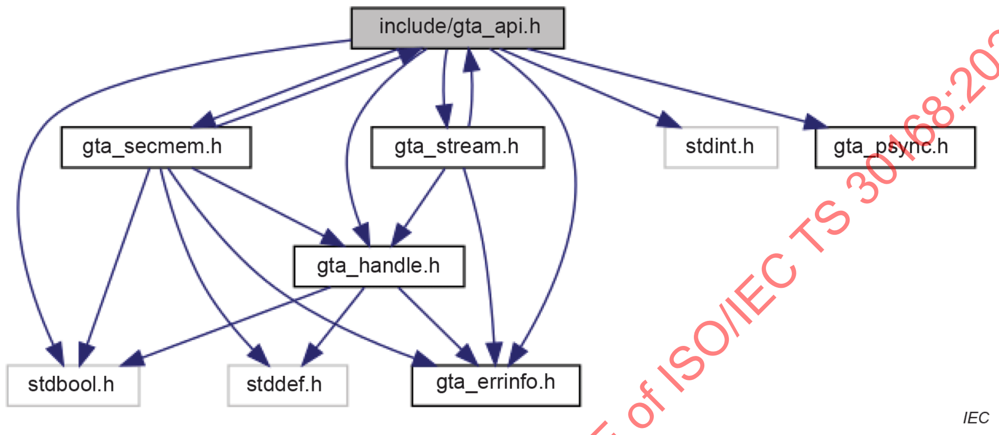


Figure A.1 – Dependency graph for `gta_api.h`

A.2 Application interface – gta_api.h

The header file gta_api.h contains all function calls that shall be provided by GTA API according to the description in Clause 6. This file also includes proxy functions for all provider functions defined in Clause A.3.

A reference header file for `gta_api.h` is provided at: https://standards.iso.org/iso-iec/ts/30168/ed-1/

A.3 Provider interface – gta_apif.h

The header file `gta_apif.h` contains all function calls that shall be implemented for individual SE providers.

A reference header file for `gta_apif.h` is provided at: https://standards.iso.org/iso-iec/ts/30168/ed-1/

A.4 Handles – gta_handle.h

The header file `gta_handle.h` contains GTA API definitions for handle management.

A reference header file for `gta_handle.h` is provided at: https://standards.iso.org/iso-iec/ts/30168/ed-1/

A.5 Function parameter I/O streams – gta_stream.h

The header file `gta_stream.h` contains GTA API definitions for I/O streams. The stream interface is used by GTA API to receive values from an application and to return results to an application. To use GTA API, the application shall provide an appropriate implementation for this interface. A vendor can include a reference implementation for a stream interface with its implementation of GTA API.

A reference header file for `gta_stream.h` is provided at:
https://standards.iso.org/iso-iec/ts/30168/ed-1/

A.6 Error information – gta_errinfo.h

The header file `gta_errinfo.h` contains GTA API definitions for error information.

A reference header file for `gta_errinfo.h` is provided at:
https://standards.iso.org/iso-iec/ts/30168/ed-1/

A.7 Secure memory management – gta_secmen.h

The header file `gta_secmem.h` contains GTA API definitions for secure memory management.

A reference header file for `gta_secmem.h` is provided at:
https://standards.iso.org/iso-iec/ts/30168/ed-1/

A.8 Process synchronization – gta_psync.h

The header file `gta_psync.h` contains GTA API definitions for process synchronization.

A reference header file for `gta_psync.h` is provided at:
https://standards.iso.org/iso-iec/ts/30168/ed-1/

Annex B
(normative)

Basic profiles

B.1 ch.iec.30168.basic.passcode

B.1.1 Description

The profile ch.iec.30168.basic.passcode allows the implementation of simple authentication methods for the local SE. The profile ch.iec.30168.basic.passcode shall be supported as a basic profile with every implementation of GTA API.

A personality representing a shared passcode can be set up between the application or user and the SE. Later, the application or user can provide the passcode to the SE in order to authenticate itself. A successful authentication yields a personality access token (cf. 5.6.5.2.4) that can be used to access privileged operations.

The same profile name ch.iec.30168.basic.passcode is used for a deployment (cf. 5.6.3.2) and a usage profile (cf. 5.6.3.4).

NOTE Passcodes according to ch.iec.30168.basic.passcode are not intended to be changed during their lifetime. In case this functionality is required in the future, a more advanced passcode profile can be defined.

B.1.2 Deployment

New passcodes are installed using gta_personality_deploy().

Table B.1 specifies the properties of a personality that is deployed with profile ch.iec.30168.basic.passcode.

Table B.1 – ch.iec.30168.basic.passcode deployment properties

Property	Description
Security Mechanism	Verification of knowledge of a secret value. Either the secret value itself or a representative of that value usable for verification can be kept within the SE. For example, a cryptographic digest of the passcode can be used as representative for a passcode.
Import Format	The passcode is provided as ‘\0’ terminated string. Valid characters are numbers (‘0’ to ‘9’) lower-case (‘a’ to ‘z’) characters, upper-case characters (‘A’ to ‘Z’), and selected special characters (‘(‘, ’)’, ‘[‘,’]’, ‘{‘,’}’,’%’, ‘*’, ‘&’, ‘-‘, ‘+’, ‘<’, ‘>’, ‘!’, ‘?’, ‘=’, ‘$’, ‘#’) from the ASCII character set. Length of a passcode according to this profile should be at least 16 bytes.
Fingerprinting	The 64 byte fingerprint is built as follows: • Starting byte gives an identifier; In case the personality fingerprint for ch.iec.30168.basic.passcode has been calculated as specified here, the byte shall be set to 0x01. • Following 32 bytes are a random 256 bit salt value • Following 7 bytes are supposed to be set to 0x00 • Following 24 bytes are a 192 bit SHA-3 hash that is calculated over the first 40 bytes concatenated with the personality name (excluding the terminating ‘\0’) and the value of the passcode (excluding the terminating ‘\0’) Fingerprint:= id | salt | SHA-3(salt | personality_name | passcode)
Attributes	N/A
Usage Info	Intended for usage with ch.iec.30168.basic.passcode

B.1.3 Usage

Table B.2 shows the usage properties for the profile ch.iec.30168.basic.passcode. Usage of the profile with any other function than those functions listed in the table shall result in GTA_ERROR_PROFILE_UNSUPPORTED.

Table B.2 – ch.iec.30168.basic.passcode usage properties

Property	Description
Profile Dependencies	Can use a personality that is deployed with ch.iec.30168.basic.passcode.
Supported Functions	**gta_verify()**
	h_ctx refers to a context opened with the respective personality and profile ch.iec.30168.basic.passcode. The claim data stream provides the passcode value as provided for gta_personality_deploy() (B.1.2). After successful completion of this function, an access token can be retrieved using gta_access_token_get_pers_derived().
Usage Attributes	N/A
Usage Artifact	N/A

B.2 ch.iec.30168.basic.local_data_integrity_only

B.2.1 Description

The profile ch.iec.30168.basic.local_data_integrity_only supports protection of data integrity for data that is kept locally on the IIoT device. The profile can be used to support scenarios as described in C.1.6. The profile ch.iec.30168.basic.local_data_integrity_only shall be supported as a basic profile with every implementation of GTA API.

The profile shall allow detection of any changes or tampering with data which has been protected. In addition, the profile binds the integrity proof to a specific IIoT device. Any attempts to verify the integrity of the data on a different IIoT device shall fail. Any data that is processed shall stay available in cleartext, that is, this profile shall not incorporate any kind of data confidentiality related cryptography.

The same profile name ch.iec.30168.basic.local_data_integrity_only is used for a creation and usage profile (cf. 5.6.3.2).

B.2.2 Creation

To use the profile, a new personality shall be created using gta_personality_create(). As the personality is only used locally, there is no corresponding enrollment profile. There can be multiple personalities using this profile on the same device, for example, to allow different applications to protect their data independently of each other.

Table B.3 specifies the properties of a personality that is created with profile ch.iec.30168.basic.local_data_integrity_only.

Table B.3 – ch.iec.30168.basic.local_data_integrity_only creation properties

Property	Description
Security Mechanism	The cryptographic mechanisms used to realize this profile are left to the decision of the SE provider depending on the capabilities of the SE being used. The mechanisms that are used shall be documented by the vendor. The provided security level should be at least equivalent to 128 bit secret key cryptography.
Fingerprinting	Fingerprint is provided as a unique 256 bit value. The fingerprint is expected to change whenever a new personality is created on the same device, even in case personality names or identifiers are reused.
Attributes	N/A
Usage Info	Intended for use with ch.iec.30168.basic.local_data_integrity_only

B.2.3 Usage

Table B.4 shows the usage properties for the profile ch.iec.30168.basic.local_data_integrity_only. Usage of the profile with any other function than those functions listed in the table shall result in GTA_ERROR_PROFILE_UNSUPPORTED.

Table B.4 – ch.iec.30168.basic.local_data_integrity_only usage properties

Property	Description
Profile Dependencies	Shall be usable with any personality that is created with ch.iec.30168.basic.local_data_integrity_only. May be usable with personalities that are created with ch.iec.30168.basic.local_data_protection.
Supported Functions	**gta_seal_data()**
	gta_seal_data() computes a protected data BLOB from the data BLOB provided as input data. The provided data BLOB gets a part of the protected data BLOB. It shall be possible to recover the original data BLOB from the protected data BLOB. Data recovery shall not depend on secret or private information used to protect the data (that is, no encryption).
	gta_unseal_data()
	gta_unseal_data() recovers the original data BLOB from a previously protected data BLOB, that is, this function provides the inverse operation to gta_seal_data(). In case the integrity of the data could not be verified, the function fails and returns an error.
	gta_authenticate_data_detached()
	gta_unseal_data_detached() computes an integrity check value from the data BLOB provided as input data.
	gta_verify_data_detached()
	gta_verify_data_detached () verifies whether the data still matches the provided integrity check value that was previously computed on the same device over the same data with gta_authenticate_data_detached().
Usage Attributes	N/A
Usage Artifact	The format of the resulting artifact (protected data BLOB or integrity check value for detached mode) is not specified as part of this document. These artifacts are only used on the local device. Vendors can either use standardized formats (for example, PKCS#7) or deliberate proprietary format. Whatever format is used, the complementary behaviour regarding gta_seal_data()/gta_unseal_data() and gta_authenticate_data_detached()/gta_verify_data_detached() shall be as described above.

B.3 ch.iec.30168.basic.local_data_protection

B.3.1 Description

The profile ch.iec.30168.basic.local_data_protection supports protection of data integrity and confidentiality for data that is kept locally on the IIoT device. The profile can be used to support scenarios as described in C.1.6. Vendors can provide the profile ch.iec.30168.basic.local_data_protection as basic profile along with an implementation of GTA API.

The profile shall allow detection of any changes or tampering with data which has been protected and shall inhibit exposure of the protected data. In addition, the profile binds the integrity proof and the ability to recover the data to a specific IIoT device. Any attempts to verify the integrity of the data or to recover the data on a different IIoT device shall fail.

The same profile name ch.iec.30168.basic.local_data_protection is used for a creation and a usage profile (5.6.3.2).

B.3.2 Creation

To use the profile, a new personality shall be installed using gta_personality_create(). As the personality is only used locally, there is no corresponding enrollment profile. There can be multiple personalities using this profile on the same device, for example, to allow different applications to protect their data independently of each other.

Table B.5 specifies the properties of a personality that is created with profile ch.iec.30168.basic.local_data_protection.

Table B.5 – ch.iec.30168.basic.local_data_protection creation properties

Property	Description
Security Mechanism	The cryptographic mechanisms used to realize this profile are left to the decision of the SE provider depending on the capabilities of the SE being used. The mechanisms that are used shall be documented by the vendor. The provided security level should be at least equivalent to 128 bit secret key cryptography.
Fingerprinting	Fingerprint is provided as a unique 256 bit value. The fingerprint is expected to change whenever a new personality is created on the same device, even in case personality names or identifiers are reused.
Attributes	N/A
Usage Info	Intended for use with ch.iec.30168.basic.local_data_protection.

B.3.3 Usage

Table B.6 shows the usage properties for the profile ch.iec.30168.basic.local_data_protection. Usage of the profile with any other function than those functions listed in the table shall result in GTA_ERROR_PROFILE_UNSUPPORTED.

Table B.6 – ch.iec.30168.basic.local_data_protection usage properties

Property	Description
Profile Dependencies	Shall be usable with any personality that is created with ch.iec.30168.basic.local_data_protection.
Supported Functions	**gta_seal_data()**
	gta_seal_data() computes a protected data BLOB from the data BLOB provided as input data. The provided data BLOB gets a part of the protected data BLOB. It shall be possible to recover the original data BLOB from the protected data BLOB. Recovery of data shall require access to the IIoT device and personality that was used to protect the data.
	gta_unseal_data()
	gta_unseal_data() recovers the original data BLOB from a previously protected data BLOB, that is, this function provides the inverse operation to gta_seal_data(). The function can be realized using two distinct mechanisms for integrity and confidentiality protection. The function can return different error codes to indicate whether the failure relates to integrity or confidentiality protection. However, regardless of the type of error no data shall be returned if there is an error.
Usage Attributes	N/A
Usage Artifact	The format of the resulting artifact (protected data BLOB or integrity check value for detached mode) is not specified as part of this document. Artifacts are only used on the local device. Vendors can either use standardized formats (for example, PKCS#7) or deliberate proprietary formats. The complementary behaviour regarding gta_seal_data() and gta_unseal_data() shall be as described above.

Annex C
(informative)

Example security scenarios for Industrial IoT

C.1 Analysis of example security scenarios for IIoT

C.1.1 General

Requirements towards the GTA API are based on the analysis of a selection of industrial scenarios for security. Subclauses C.1.2 to C.1.10 provide an overview of example scenarios that are considered.

C.1.2 Scenarios for application protocols

C.1.2.1 OPC UA

C.1.2.1.1 Overview

OPC UA is a platform-independent standard through which various kinds of systems and devices can communicate. Communicatoin is done by sending request and response messages between clients and servers or network messages between Publishers and Subscribers. Various types of networks can be used with OPC UA. It supports robust, secure communication that assures the identity of OPC UA applications and resists attacks.

In the client–server model OPC UA defines sets of services that servers can provide, and individual servers specify to clients what Service sets they support. Information is conveyed using OPC UA-defined and vendor-defined datatypes, and servers define object models that Clients can dynamically discover. Servers can provide access to both current and historical data, as well as Alarms and Events to notify clients of important changes. OPC UA can be mapped onto various of communication protocols. Data transferred via OPC UA can be encoded in various ways to trade off portability and efficiency.

In addition to the client–server model, OPC UA defines a mechanism for Publishers to transfer information to Subscribers using the PubSub model.

OPC UA is applicable to components in all industrial domains. Example components are, industrial sensors and actuators, control systems, Manufacturing Execution Systems and Enterprise Resource Planning Systems, including IIoT, M2M as well as Industrie 4.0. These systems are intended to exchange information and to use command and control for industrial processes. OPC UA defines a common infrastructure model to facilitate this information exchange. OPC UA specifies the following:

- the information model to represent structure, behaviour, and semantics;
- the message model to interact between applications;
- the communication model to transfer the data between endpoints;
- the conformance model to guarantee interoperability between systems.

Subclause C.1.2 describes scenarios for OPC UA based on the services that are defined in the OPC UA Server Specification, Part 4 [5]. There are a lot of OPC UA services which do not have any requirements for security. This analysis mentions only those services which possibly impact the GTA API.

C.1.2.1.2 Scenario overview

For OPC UA, the Part 4 [5] provides an overview and a detailed description of the OPC UA services and the client/server behaviour. According to this documentation, Table C.1 summarizes the scenarios which can be derived from the OPC UA services.

Table C.1 – Scenarios for OPC UA client and server

Reference	Scenario description
OPC#1 OPC#Discovery-Services	An OPC UA client wants to • find OPC UA servers in its environment. The contacted Discovery Server returns the known OPC UA servers and known Discovery Servers. This service can be used without security; but can also be used over a secure channel. • explore OPC UA service endpoints on a dedicated OPC UA server. The contacted OPC UA server returns the supported endpoints and all the configuration information that is required to establish a Secure Channel and a Session. • register another OPC UA server at the Discovery Server. The corresponding services are not used by OPC UA clients. The main security focus for the OPC UA server and the client is to provide and use the corresponding services in a way that integrity and confidentiality of the transmitted data are preserved. Thus, the security mechanisms that are needed for this service are provided by the mechanisms that are needed for opening and closing a secure channel.
OPC#2 OPC#Open-Secure-Channel	An OPC UA client wants to set (or renew) a secure channel to a different endpoint (for example, an OPC UA server). The corresponding endpoint establishes (if authentication succeeds) this channel with the client. Each secure channel has a globally unique identifier and is valid for a specific combination of client and server OPC UA application instances. It contains one or more security tokens that identify cryptographic keys that are used to encrypt and authenticate messages. Tokens usually have a limited lifetime. The secure channel is assigned a security mode which can be "None", "SignOnly", or "SignAndEncrypt".
OPC#3 OPC#Close-Secure-Channel	An OPC UA client terminates a secure channel. The request is signed by the corresponding token that is used for the channel (usually the Application Instance Certificate). If verification succeeds, the channel endpoint terminates the secure channel.
OPC#4 OPC#Create-Session	An OPC UA client wants to create a session over a secure channel using the corresponding services. The OPC UA server returns two values which uniquely identify the session. One value holds the session ID which is used to identify the session. The other value holds the authentication token which is used to associate an incoming request with a session (the token can be a long random number). The OPC UA client also sends a nonce which is signed by the server and then verified by the client. In this way, the server shows Proof-of-Possession of its instance certificate. In the response, the server sends its nonce which is signed by the client to show Proof-of-Possession of its application instance certificate.
OPC#5 OPC#Activate-Session	An OPC UA client wants to activate a session which has been set up before. The corresponding services are used by the client to specify the identity of the user associated with the session. The request is issued by the client before any further service request, except the "Close Session" service. The OPC UA server verifies the signed server nonce, in this way the client can perform Proof-of-Possession of its application instance certificate. The client authenticates to the server by providing a user identity in form of an identity token. There are several possibilities for the client authentication: If the token is a "UserNameIdentityToken", authentication is done by a password that is included in the token. If the token is an X509IdentityToken, authentication is done by a signature. Further possibilities include Kerberos tokens or OAuth2 tokens.

Reference	Scenario description
OPC#6 OPC#Data-Services	An OPC UA client wants to • manage and organize OPC UA data on an OPC UA server using corresponding services. All services the client can use for these scenarios only require an existing secure channel. • browse and view OPC UA data on an OPC UA server. The corresponding services allow clients to discover nodes in a view by browsing in data provided by an OPC UA server. Browsing allows clients to navigate up and down the hierarchy, or to follow references between nodes contained in the view. In this manner, browsing also allows clients to discover the structure of the view. The services are recommended to be used over a secure channel. • query OPC UA data from an OPC UA server. The Query Services allow clients to access the server address space without browsing and without knowledge of internal storage of the data. Querying allows clients to select a subset of the nodes in a view based on some client-provided filter criteria. The nodes that are selected from the view by the query statement are called a result set. The services are recommended to be used over a secure channel. The main security focus for the OPC UA server and the client is to provide and use the corresponding services in a way that integrity and confidentiality of the transmitted data are preserved. Thus, the security mechanisms that are needed for this service are provided by the mechanisms that are needed for opening and closing a secure channel.
OPC#7 OPC#Attribute-Services	An OPC UA client wants to access attributes on an OPC UA server that are part of nodes. The corresponding attribute services can be used by the client to read and write attribute values. Attributes are primitive characteristics of nodes that are defined by OPC UA. The services are recommended to be used over a secure channel. The main security focus for the OPC UA server and the client is to provide and use the corresponding services in a way that integrity and confidentiality of the transmitted data are preserved. Thus, the security mechanisms that are needed for this service are provided by the mechanisms that are needed for opening and closing a secure channel.
OPC#8 OPC#Method-Services	An OPC UA client wants to use methods provided by objects on an OPC UA server. Methods represent the function calls of objects. They are invoked and return after completion, whether successful or unsuccessful. The corresponding services define the means to invoke methods. Discovery of methods is provided through the browse and query services. Clients discover the methods that are supported by an OPC UA server by browsing for the owning objects that identify their supported methods. The services are recommended to be used over a secure channel. The main security focus for the OPC UA server and the client is to provide and use the corresponding services in a way that integrity and confidentiality of the transmitted data are preserved. Thus, the security mechanisms that are needed for this service are provided by the mechanisms that are needed for opening and closing a secure channel.

Reference	Scenario description
OPC#9 OPC#PubSub-Services	An OPC UA client wants to • subscribe to data or events that are provided by an OPC UA server by defining Monitored Items. Each Monitored Item identifies the item to be monitored and the subscription to be used to send notifications. The item to be monitored can be any Node Attribute. The monitoring can be influenced by different parameters that tell the OPC UA server how the item is to be sampled, evaluated, and reported. These parameters are the sampling interval, the monitoring mode, the filter, and the queue parameter. The services are used over a secure channel. • be notified if a certain event (for example, change of a node value) occurs on an OPC UA server. Subscriptions are used to report notifications to the client. Subscriptions include Monitored Items that are assigned to them by the client which generate notifications that are reported to the client. The services are used over a secure channel. The main security focus for the OPC UA server and the client is to provide and use the corresponding services in a way that integrity and confidentiality of the transmitted data are preserved. Thus, the security mechanisms that are needed for this service are provided by the mechanisms that are needed for opening and closing a secure channel.

C.1.2.2 PROFINET security extensions

C.1.2.2.1 Overview

Industrial communication using protocols such as PROFINET is growing in importance. Ensuring security will be viewed as the key requirement for future automation solutions. For this reason, the PROFINET CB/WG 10 Security PI working group set focus on this topic.

In the current version, as of the time of this writing, PROFINET does not yet provide any security features on the protocol level. The Security PI working group already published the whitepaper "Security Extensions for PROFINET" [1]. The whitepaper provides an initial look at the planned security extensions for the protocol.

The document starts with a presentation of the general security objectives from a general point of view. This presentation is supplemented by a brief assessment of the relevance for PROFINET:

- Integrity: Relevance "High"
- Confidentiality: Relevance "Low"
- Availability: Relevance "High"
- Authenticity: Relevance "High"
- Authorization: Relevance "High"
- Non-repudiation: Relevance "Medium"

In a next step, the generic security objectives are mapped to PROFINET-specific security objectives. Finally, a basic concept for PROFINET protocol security is presented. This concept is based on public key certificates.

As it has turned out, the security objectives provide varying priorities, especially for the aspect of confidentiality of data. For this reason, three PROFINET security classes for IIoT security were introduced in [1] and are listed in Table C.2.

Table C.2 – Security classes for the PROFINET protocol

Security class	Name	Definition
1a	Robustness	Basic protection including read-only discovery and configuration protocol (DCP), SNMP configuration, and integrity protection for general station description (GSD) files
2	Integrity + Authenticity	Protection of the authenticity and integrity of information exchanged over a PROFINET communication relation. Class 1a is a prerequisite for class 2
3	Confidentiality	Protection of the confidentiality of information exchanged over a PROFINET communication relation. Class 2 is a prerequisite for class 3.

C.1.2.2.2 Scenario overview

Reference [1] already presents specific security requirements, such as the requirements used in the definition of the different PROFINET security classes.

Nevertheless, for the sake of completeness the more general scenarios for PROFINET with security extensions are briefly presented in Table C.3. These scenarios are derived from the information that is given in [1].

Table C.3 – Scenarios for PROFINET security

Reference	Scenario description
PRN#1 PRN#Data-Integrity	The user of a system wants to be ensured that the integrity and authenticity of general data are protected. Examples for general data are configuration data and GSD files. This applies to general data in transit and at rest.
PRN#2 PRN#Data-Protection	The user of a system wants to be ensured that the integrity, authenticity and confidentiality of general data are protected. Examples for general data are configuration data and GSD files. This applies to general data in transit and at rest.
PRN#3 PRN#IO-Data-Integrity	Realtime of I/O data is an important aspect of PROFINET. This scenario is scenario PRN#Data-Integrity regarding realtime requirements.
PRN#4 PRN#IO-Data-Protection	Realtime of I/O data is an important aspect of PROFINET. This scenario is scenario PRN#Data-Protection regarding realtime requirements.
PRN#5 PRN#Time-Integrity	The manufacturer wants to ensure that the integrity of the clock synchronization is given.
PRN#6 PRN#Firmware-Integrity	The manufacturer wants to ensure that the devices run only the original firmware.
PRN#7 PRN#Secure-Identity	The user wants to ensure that each device, controller and supervisor provides a unique identity, which cannot be impersonated by others.
PRN#8 PRN#Secure-Identity-User	A user of a system wants to be ensured that no other instance can use services of the system with the account information of the user.
PRN#9 PRN#Mutual-Authentication	A communication peer – for example, device, controller, or supervisor – wants to be ensured that the communication partner cannot pretend a faked identity.

C.1.2.3 Secure communication

C.1.2.3.1 Overview

A secure communication protocol provides security services for traffic at a specific network layer. The security services include, for example, authenticity and integrity of data that is transported between two peers. Data confidentiality and entity authentication of the peers are important scenarios for secure communication services.

Widely deployed secure communication protocols are, for example, TLS [6] and datagram TLS (DTLS) [7] for the transport layer, IPsec [8] for the network layer, and SSH [9] for the application layer.

The future PROFINET version including security extension as mentioned in C.1.2.2 is another example of a secure communication protocol.

C.1.2.3.2 Scenario overview

The security requirements of a secure communication protocol are straightforward and can be derived from the protocol specification itself. For the sake of completeness, the more general scenarios for a secure communication protocol are briefly presented in Table C.4.

Table C.4 – Scenarios for secure communication protocols

Reference	Scenario description
COM#1 COM#Entity-Authentication	Communication peer A wants to verify the presented identity of peer B. It is expected not to be possible that another peer impersonates the identity of peer B.
COM#2 COM#Data-Integrity	Communication peer A wants to be ensured that data received from another peer was not modified or replayed by others during transportation.
COM#3 COM#Data-Authenticity	Communication peer A wants to be ensured that data received supposedly from peer B was also originally sent by peer B.
COM#4 COM#Data-Confidentiality	Communication peer A wants to be ensured that no eavesdropper can read the content of the data that is sent to peer B.
COM#5 COM#Non-Repudiation	Communication peer A wants to be ensured that peer B cannot deny the origin of a message peer B sent to peer A.

C.1.3 Secure device identities

C.1.3.1 Overview

Different ecosystems have disparate needs and practices. It is clear that automation and zero touch mass registration are necessary for scaling up to billions of devices. A secure identity (anchored in hardware) is at the centre.

Secure identities are a prerequisite for many other protection measures. They are important for every scenario which aims at uniquely identifying a device and binding certain actions or permissions to it. In the digital world, it is necessary to verify which device has been granted access to data or has used certain services in automation system.

Secure identities are also important for legal and commercial processes. In principle, digital identities increase the transparency of processes. Transparency ensures the understanding who, how, when and with what rights someone communicates and possibly decides.

Identities also play a key role for the functions of device authentication and agreeing on keys for communication protection, like IPsec, TLS or other protocols.

IEEE 802.1AR [10] is a standard that can be used for device identities. According to IEEE 802.1AR, the device identity is an IDevID that includes a unique device identifier. The IDevID is generated or set during manufacturing and used to securely download local device identifiers (LDevIDs) used for services like authentication, communication, or management.

Table C.5 summarizes the most important scenarios for IIoT devices that are used in Industrie 4.0 IACS scenarios.

C.1.3.2 Scenario overview

According to C.1.3.1, the scenarios considering "secure identities" are summarized in Table C.5.

Table C.5 – Scenarios for secure identities

Reference	Scenario description
SID#1 SID#Unique-Identification	The asset owners want their IIoT devices to be capable of uniquely identifying themselves to other devices. The identification can either be non-trustworthy (performed non-cryptographically) or trustworthy by cryptographic support (see also scenarios "proof of origin" or "device authentication").
SID#2 SID#Proof-of-Origin	The asset owners want their IIoT devices which integrate a trust anchor to be capable of proving that they have a correct manufacturer origin. Usually, to implement this scenario, a manufacturer device certificate (MDC) is provisioned and securely stored in the IIoT device. The proof of origin is done by verifying the certificate and performing proof-of-possession of the corresponding private key. Alternatively, a symmetric approach can be chosen using an appropriate key management.
SID#3 SID#Device-Authentication	The asset owners want their IIoT devices to be capable proving their identity to another device, that is, to authenticate themselves to other devices. The solution alternatives for this scenario are equivalent to the scenario "proof of origin".
SID#4 SID#M2M-Authentication	This scenario is equivalent to the scenario "device authentication". Nevertheless, it is mentioned here to emphasize the authentication aspect where no human user is present, but all security mechanisms are solely based on the IIoT device.
SID#5 SID#Secure-Binding-to-Engineering	The user of an engineering tool wants to ensure that the engineering tool is communicating to a legitimate or licensed field device. The field device performs some form of identification and authentication to the engineering tool, see also scenarios "proof of origin" or "device authentication".

C.1.4 Supply-chain and trustworthiness/authenticity of device

C.1.4.1 Overview

Having trust in IIoT devices means having trust in the device manufacturing process:

- being manufactured only using trusted components behaving in the expected way;
- being manufactured in the expected way and by the expected manufacturer;
- being manufactured in a robust way.

The first aspect considers the supply chain of all components by which the device is assembled. Security of the supply chain is needed to fulfil the expectations of predictable device behaviour. This expectation extends naturally to the process of assembling the device using the different components. Furthermore, it is expected to be difficult for an attacker to subvert a device by replacing certain components. Likewise, an attacker should also be prevented from copying the complete device and reusing or reselling this device instead of the original trustworthy device. As a result, these expectations mean that some way to check the authenticity of a device is required.

C.1.4.2 Scenario overview

According to C.1.4.1, the scenarios listed in Table C.6 are identified for the topic "Trustworthiness of devices":

Table C.6 – Scenarios for device trustworthiness

Reference	Scenario description
TWD#1 TWD#Anti-Counterfeiting	The IIoT asset (device) owners want to protect their IACS solutions against fraudulent devices. A mechanism is implemented in the devices which identifies them as genuine and in such a way ensures trustworthiness of the installed devices. To achieve this goal, usually a trusted component is brought into the devices. This trusted component (for example, a dedicated security controller) performs device authentication. Device authentication establishes the trust into the genuineness of the whole device by being bound to the device. For example, device binding can be implemented by gathering information from each component and concatenating this information into the authentication process.
TWD#2 TWD#Supply-Chain-Security	The IIoT asset owners want to ensure that all components in their IIoT devices are genuine and manufactured and assembled in the expected way. For achieving this goal, the device owner collects information about the overall device assembling process. Each component is expected to provide explicit protection against counterfeiting, or trustworthiness of its supply chain is expected to be established at least implicitly.
TWD#3 TWD#Cloning-Prevention	The manufacturers want to protect their devices against cloning. Protection against fraudulent devices is not the main security focus of this scenario, but the protection against damage of the manufacturer's business model. Also the asset owner has a large interest to be protected against cloned devices. The asset owner loses warranty if the manufacturer detects that the device has not been manufactured by themselves.
TWD#4 TWD#Device-Robustness	Protection of the device extends beyond the manufacturing process. The asset owners also want protection of their devices against manipulation after installation in the IACS system. Protection mechanisms can be divided into two mechanisms: Mechanisms considering the protection of software on the device (see also C.1.5) and mechanisms considering the manipulation/tampering of the device itself. Thus, the asset owners want to add one or more of tamper prevention, tamper detection and tamper reaction mechanisms in their IACS systems and their devices. Anti-tamper mechanisms prevent the devices from being manipulated in the production environment.

C.1.5 Device integrity protection

C.1.5.1 Overview

System integrity regarding IIoT and industrial applications means that there is assurance that the system runs only authentic firmware and software. Authentic firmware and software originates from the original manufacturer, the system vendor, and the end user. Further, it is important that the system is based on the software and hardware settings that are predefined and selected by the manufacturer or vendor. If also needed, measures can be taken to detect the usage of nonconformant hardware and components. Finally, the system should also be able to detect and to log incidents that can violate the integrity of the system.

System integrity attacks can roughly be classified into static and dynamic attacks. In a static attack, the attacker has full access to the system and components, but the attacker is not able to start or to emulate the system. Whereas in a dynamic attack, the attacker can also start and attack the system during normal execution. Buffer overflow attacks and reverse engineering are examples for a system integrity violation during normal execution.

System integrity protection is a crucial aspect especially for critical infrastructure systems and medical devices.

C.1.5.2 Scenario overview

According to the more general introduction in C.1.5.1, the scenarios that are listed in Table C.7 are identified concerning system integrity protection.

Table C.7 – Scenarios for system integrity protection

Reference	Scenario description
SIP#1 SIP#Secure-Boot-Chain	The manufacturer wants to ensure that the devices only boot the original boot chain, the original operating system and the original system modules and software/hardware settings.
SIP#2 SIP#Secure-Software-Update	The manufacturer wants to ensure that the system accepts only original and up-to-date software updates that are provided by an authorized instance.
SIP#3 SIP#Runtime-Monitoring	The user or administrator wants to monitor the system state at runtime for incidents to detect potential system integrity violations as soon as possible.
SIP#4 SIP#Runtime-Integrity-Checks	The user or administrator wants the system to perform runtime integrity checks to detect and to report system integrity violations that happened during normal execution.
SIP#5 SIP#Remote-Attestation	A remote party wants to verify if the system integrity of a system that the remote party intends to interact with is a certain security level or not. Interaction means usage of the system by the remote party or communication to the system by the remote party.
SIP#6 SIP#Secure-Logging	The user or administrator wants to ensure that only the system can log integrity violations. It is expected to be unfeasible for an attacker to fake or to delete log entries.
SIP#7 SIP#File-System-Integrity	The user or administrator wants to ensure that the system protects the used file systems from unauthorized modifications.
SIP#8 SIP#Protection-Against-Device-Manipulation	The user wants to ensure that the system detects and reacts to the usage of nonconformant hardware and hardware components.

C.1.6 Application security

C.1.6.1 Overview

The scenarios within this category focus on the protection of the confidentiality of data (the integrity of data has been investigated and described in C.1.5). This data can be dynamic data that is generated by an application, configuration data for an application or the application itself. Often the access to user application-specific data is restricted by a mechanism that is known from the ARM TrustZone approach. This mechanism is summarized by the notion "Secure Execution environment (for user applications)".

Protection of an application itself can, for example, comprise the firmware of the supplier or a dedicated user application. Protecting these assets is also known by the notion "know-how-protection" or "IP protection" (IP stands for intellectual property). Intellectual property is often one of the most valuable assets of a software developer or a software company. If an attacker or a competitor gets knowledge about the internal functionality of (parts of) the software or the proprietary algorithms, someone could copy or adapt those algorithms for personal needs and resell the software. In this way, the business model of the original provider could be severely damaged.

Thus, it is essential, that the expertise and knowledge residing inside the algorithms of the proprietary software are protected against different forms of analysis. One possible approach for an attacker is static analysis. For static analysis, no further prerequisites are assumed except (public) availability of the software the attacker wants to analyse or access to the device on which the corresponding software resides. A different approach is dynamic analysis. Dynamic analysis presupposes that the attacker has access to a device on which the software runs, and also needs the capability to analyse the running system (for example, to extract memory or run a debugger).

If the software itself cannot be protected, there are also scenarios which rely on adequate measures of the overall system supporting the protection against extraction of knowledge and expertise that is represented by software. Protection of the software itself can also be supported by these additional measures of the system.

C.1.6.2 Scenario overview

Table C.8 summarizes scenarios corresponding to the protection against extraction of knowledge and experience that is represented by proprietary software.

Table C.8 – Scenarios for know-how protection

Reference	Scenario description
APS#1 APS#Protection-of-Volatile-Application-Data	The user of an application (for example, device owner, user) wants to protect volatile application data against read-out by an illegitimate person or device. This volatile data is only available during the runtime of the application. An attacker needs access to the running device to perform this attack. A possible approach is to apply RAM encryption. With RAM encryption all volatile data only is available in encrypted form. Nevertheless, at some point where this data is processed, it is available in decrypted form. Any approach for hardening the access and read-out of volatile memory, for example, closing of debug ports, also complicates this kind of attack.
APS#2 APS#Protection-of-Persistent-Application-Data	The user of an application (for example, device owner, user) wants to protect the data which is necessary for application configuration or which is output and stored by the application. This data is protected against read-out by an illegitimate person or device. The attacker only needs access to the device (or the relevant data storage media). If the attacker has access to the data storage in some way, the common approach for data protection is encryption. It is important that encryption is combined with appropriate key management. The data is encrypted before it is persistently stored and decrypted before it is reused.
APS#3 APS#Secure-Execution-Environment	This scenario is equivalent to the latter scenario. Users of an application (for example, device owner, user) want to protect the data that is used or output by their applications from illegitimate read-out. Encryption of data can be enhanced or replaced by restricting access to the wanted data. Access restriction can be realized through usage of a "Secure Execution Environment", sometimes also called "Secure World". The users load their application into the "Secure World". The security mechanisms for accessing this secure environment allow access to the application and its data using a dedicated interface and using cryptographic authentication. Circumvention of the specified interfaces to access the application or corresponding data requires enhanced attacker capabilities.
APS#4 APS#Protection-against-Static Analysis	The asset owners want critical software/algorithms on their IIoT devices to be protected against static code analysis. The attacker is no longer able to use static analysis to learn internals about the software (for example, internal functionality of the algorithms, parameterization of algorithms, control flow of the software). The attacker has access to the device but cannot run the software on the device. It is possible to extract the software from storage components and statically analyse the binary code from which an attacker wants to extract knowledge.

Reference	Scenario description
APS#5 APS#Protection-against-Dynamic-Analysis	The asset owners want that critical software/algorithms on their IIoT devices are protected against dynamic code analysis. The attacker is no longer able to use dynamic code analysis to learn internals about the software (for example, internal functionality of the algorithms, parameterization of algorithms, control flow of the software). The attacker has access to a device which runs the software under attack. This device can be a dedicated device in the IACS environment but can also be a device or an emulation in the hand of the attacker. The attacker wants to extract know-how from the running software.
APS#6 APS#File-System-Encryption	The asset owners want their IIoT devices to be protected against attackers which run offline attacks on persistent data that is stored on the device. The attacker is prevented from, for example, extracting knowledge from binaries or exporting sensitive data. Those attacks are especially interesting after separation of the physical storage device from the device (for example, maintenance, disposal). To achieve this goal, the complete file system is encrypted. Decryption of the file system is only possible after a pre-defined set of conditions is met. Pre-conditions can be set in a way that decryption is only possible if the overall device integrity (physical and logical) is present.
APS#7 APS#Debug-Protection	The asset owners want their IIoT devices to be protected against attackers which dynamically analyse the firmware or software components running on their IIoT devices. The asset owner wants to prevent an attacker from connecting to a debug port and stepping through the running code. Anti-debug protection measures are enabled which prevent an attacker from analysing the running software on the IACS component.

C.1.7 Feature licensing

C.1.7.1 Overview

Licensing of software features or hardware features is an important aspect to maximize the vendors monetization of software and hardware products. And even for industry-specific applications and services, the importance of feature licensing is increasing.

Often distributed software includes all available features, fully implemented, when shipped to or when downloaded by the end customer. Even if the full software is exposed to the end customer, access is to be restricted to those features for which a license has been purchased. Hence, this situation poses many security concerns.

The approach that is chosen to implement a feature licensing system depends on various factors, for example:

- the connectivity of the licensee;
- the underlying specific license models;
- the existing incentives to attack the licensing system;
- the technical skills and available resources of a potential attacker.

For example, if a license system is cloud-based, permanent connectivity between licensee and licenser is required. A license system can also be based on the usage of specific cryptographic hardware token. If permanent connectivity is not given or if sophisticated hacker attacks are expected, it is better to rely on a cryptographic token.

C.1.7.2 Scenario overview

Because license systems could be implemented in different ways, only the more general scenarios are identified and listed in Table C.9:

Table C.9 – Scenarios for feature licensing

Reference	Scenario description
FEL#1 FEL#License-Creation	The licensor wants to be ensured that only authorized instances can create and can issue a valid license. It is expected to be infeasible for an attacker to create a license which is checked as valid by the license system.
FEL#2 FEL#License-Activation	It can be preferable to differentiate between license creation and license activation. The license creation only creates a valid but still inactive license, ready for shipping to the license user. To activate the license, an additional license activation step is required. The licensor wants to be ensured that only authorized instances can perform the activation step for a valid license. It is expected to be infeasible for an attacker to perform this activation step.
FEL#3 FEL#License-Verification	The licensor wants to be ensured that the instance that finally uses a license is technically able to verify the license. License verification includes checking the integrity, validity, and the origin of licenses.
FEL#4 FEL#License-Copy-Protection	The licensor wants to be ensured that it is not feasible to copy or to duplicate license information. Unauthorized copies or duplicates of licenses are not verified as valid by the license system.
FEL#5 FEL#License-Bypass-Protection	The licensor wants to be ensured that it is hard for an attacker to remove or to bypass check points inside the application software. Check points are code where the request and the verification of a required license is finally performed by the license system.

C.1.8 Device and machine management

C.1.8.1 Overview

Correctly configured, managed, and maintained devices are essential for protecting the assets of an organization. Outdated software and unmanaged devices can leave an organization open to cyber threats.

Thus, besides the functionality of the device and the software which is installed on the device, also maintaining, updating, and other device-specific services need security. This requirement extends from beginning of the lifecycle of the device until decommissioning. After end-of-life no secrets are expected to be left on the device that could be interesting for analysis of communication with or processes running on the device.

Subclause C.1.8 summarizes all scenarios which do not have application-specific functionality in their focus, but instead the management of the device. Figure C.1 provides an overview of the individual domains (device manufacturer, machine builder, operator) and device identities (IDevIDs and LDevIDs) involved in the chosen example.

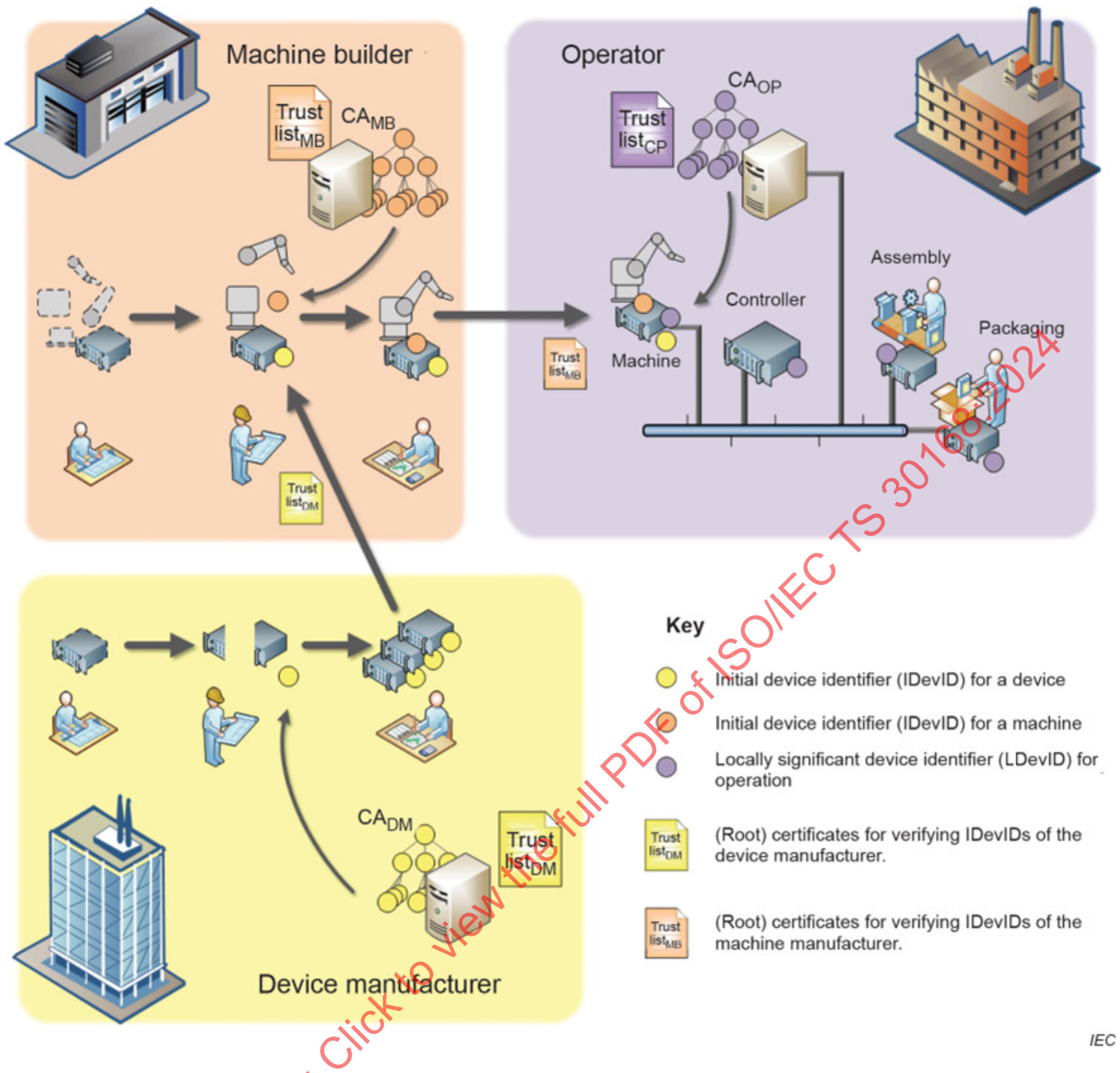

Machine builder
Trust list$_{MB}$
CA$_{MB}$
Trust list$_{DM}$
Operator
CA$_{OP}$
Trust list$_{CP}$
Controller
Assembly
Packaging
Machine
Trust list$_{MB}$
Device manufacturer
CA$_{DM}$
Trust list$_{DM}$
Key
Initial device identifier (IDevID) for a device
Initial device identifier (IDevID) for a machine
Locally significant device identifier (LDevID) for operation
(Root) certificates for verifying IDevIDs of the device manufacturer.
(Root) certificates for verifying IDevIDs of the machine manufacturer.
IEC

Figure C.1 – Device management

The device manufacturer assigns a secure initial device identity to a device at the time of production according to IEEE 802.1AR [10] (initial device identifier, IDevID). The private key can be protected by an SE within the device. The device manufacturer runs an own certification authority (CA) that is used to issue the device identity public key certificates. The basis software and configuration that are required to operate the device are installed (firmware). The SE is used to measure the integrity of the firmware during start-up. If there is an invalid measurement the device switches into an error state. Only if all measurements are correct does the device switch into full operational mode. As part of the basis configuration, trust lists containing root certificates of the device manufacturer are deployed on the device. One dedicated trust list is used to verify firmware updates signed by the device manufacturer.

The machine builder assembles several devices into a machine. The individual devices already carry the IDevID assigned by the device manufacturer. Upon receipt or before assembly of the device, the machine builder verifies the genuineness (and integrity) of the devices. This verification relies on the IDevID and a trust list containing the root certificates of the device manufacturer. This trust list is provided to the machine builder using an independent trusted channel. During assembly, the machine builder installs an application and application configuration according to the intended use of the machine. As part of the assembly process the machine builder designates one or several devices to represent the machine. The respective devices receive an additional individual IDevID assigned to the machine. This identity can either use a new private key or the existing key corresponding to the IDevID assigned by the device manufacturer. In any case, the private key is expected to be protected by the SE of the device. The machine builder runs a CA to issue certificates for the IDevIDs assigned by the machine builder. During start-up, the devices now use the SE to perform measurements. Both the firmware that is installed by the device manufacturer and the application software that is provided by the machine builder are measured. The application configuration contains trust lists of the machine builder. One dedicated trust list is used to verify software updates signed by the machine builder. Another trust list can be used to authenticate the machine builder for access for remote maintenance.

Finally, the operator installs machines in the manufacturing facility. The machines carry the IDevID assigned by the machine builder. During installation or commissioning the operator verifies the genuineness (and integrity) of the machines using the IDevID assigned by the machine builder. The trust list of the machine builder is provided over a separate trusted channel. The operator assigns as least one local identity according to IEEE 802.1AR [10] (locally significant device identifier, LDevID). This LDevID allows the machine to participate and communicate within the communication network of the manufacturing facility. The private key corresponding with the LDevID is protected using the SE of the device selected to represent the machine. As part of the deployment process the machine is configured for use within its operational environment. The configuration contains a set of trust lists containing root certificates of the operator. This particularly includes a trust list that is used to verify communication partners within the facilities network. These operator-specific trust lists are included into the measurement of the device representing the machine during start-up. The measurement is supported by the device's SE.

C.1.8.2 Scenario overview

Table C.10 summarizes all scenarios which focus on the secure device management.

Table C.10 – Scenarios for device management

Reference	Scenario description
DMG#1 DMG#Lifecycle-Provisioning	The asset owner wants to provide keys and configuration parameters for the device to be able to securely bootstrap the device for later operational use. A mechanism is necessary for manual or automatic provisioning of keys or other data on the device. This mechanism takes place during the manufacturer process or during first start-up of the device.
DMG#2 DMG#Lifecycle-Update	The asset owner wants to securely update or revoke keys. Update and revocation is required if cryptographic algorithms are broken, if there is a need for increased key length, or if certificates are compromised.
DMG#3 DMG#Lifecycle-Decommissioning	The asset owner wants to securely delete all secret and trusted data on the device if the device is decommissioned. This process will prevent attackers or unauthorized personnel from extracting secret data after end-of-life from the device.
DMG#4 DMG#Remote-Device-Maintenance	The asset owners want to maintain their device remotely to avoid having to access the device physically where it is deployed in the IACS system. For example, the asset owner wants to get data from the device, reconfigure it or install software remotely. The main security focus for the asset owner is to restrict access to the device to legitimate device administrators. It is equally important for the asset owner to avoid manipulation or read access for the data in transit.

Reference	Scenario description
DMG#5 DMG#User-Access-Control	The asset owner wants to avoid access to the device for illegitimate persons and to control access for legitimate personnel. That means, every user is granted individual access rights on the device. To achieve this goal, the asset owner wants to implement user access control mechanism. The access control mechanism enforces that no user has rights which are not really needed for the task of the user (in order to follow the least-privilege principle).
DMG#6 DMG#Device Firmware Update	Machine builder or operator wants to install a firmware update provided by the device manufacturer on the device. The criteria for measurement at boot are adjusted accordingly without affecting the application or configuration that is already installed by the machine builder or operator. It is important to ensure that only the firmware update of the device manufacturer is able the change the firmware measurements.
DMG#7 DMG#Machine Application Update	The application or configuration that is installed by the machine builder on the devices that are used to build the machine are updated. The criteria for measurement at boot are adjusted accordingly without affecting the firmware of the device manufacturer or configuration that is installed by the operator. It is important that only an application update of the machine builder can change the configuration and application measurements of the machine builder.
DMG#8 DMG#Operator Configuration Update	The operators want to change their configuration. The criteria for measurement at boot are adjusted accordingly. It is essential that this adjustment does not affect the verification of the firmware of the device manufacturer or applications and configuration that is installed by the machine builder.
DMG#9 Machine Builder Device Reuse	The machine builders want to remove their applications, configurations, and where appropriate the configuration of the operator. This process allows to reuse the device in another machine or to securely dispose of the device.
DMG#10 Machine Integrity Assurance	Machine builder wants to ensure that the machine is composed of legitimate devices. For example, an LDevID assigned to a machine can only be considered valid if the underlying IDevIDs are consistent. This consistency can be achieved by a concerted use of the SEs of the devices.

C.1.9 Blockchain/distributed ledger technology

C.1.9.1 Overview

Based on previous work by cryptographers and further research, the new millennium saw the rise of several new cryptocurrencies with Bitcoin being their most prominent member. Soon the technology found other use cases and developed further, for example, into the direction of smart contracts or distributed ledgers. Especially the latter is sometimes simplified called Blockchain. According to Wikipedia, a blockchain is "a growing list of records, called blocks, that are linked using cryptography". This link naturally has security requirements that can be supported by the GTA API. A distributed ledger is understood here as a database that is shared among a certain group of users. The participants agree on some consensus mechanism for adding entries in a way that is not reversible.

The open source Hyperledger Fabric framework is a widely known example. Subclause C.1.9 uses this framework to describe security scenarios and requirements in the realm of blockchain or distributed ledger technology. Details on Hyperledger Fabric can be found in [11]. Preservation of privacy is supported by concepts like private transactions or confidential contracts.

As one foundation for the specific security scenarios considered in the following, the Hyperledger Fabric Blockchain Crypto Service Provider (BCCSP) has been used. The BCCSP is a way to delegate certain cryptographic operations to a hardware security module (HSM). These cryptographic operations are required in the context of operations that are to be done by Fabric nodes, during their operational lifecycle phase. It essentially uses PKCS#11 [18] to interface with HSMs.

As further foundation, the Hyperledger Fabric Operations Guides and Architecture Reference [14] have been considered.

C.1.9.2 Scenario overview

Table C.11 summarizes the scenarios which focus on support of cryptographic security within blockchain or distributed ledger technology as implemented or planned by Hyperledger Fabric BCCSP according to [12] and its GitHub repository [13]. This summary is extended by further cryptographic services according to [14], whereby Membership Service Provider Implementation with Identity Mixer is not considered here.

Table C.11 – Scenarios for blockchain/distributed ledger technology (DLT)

Reference	Scenario description
DLT#1 DLT#KeyLifecycle	This scenario is expected to allow cryptographic keys to be deployed within other blockchain/distributed ledger scenarios (cf. below). Keys or key pairs, to be used within cryptographic schemes (symmetric or asymmetric) are generated or imported. Derivation of keys is also possible or public key export based on subject key identifiers. All scenarios allow several options to be set for flexibility reasons. The scenario maps to the BCCSP interface commands GenKey, DeriveKey, GetKey, and ImportKey. NOTE As of November 2019, DeriveKey is not implemented by [13].
DLT#2 DLT#Sign/Verify	This scenario allows a Fabric node to sign or verify, for example, transactions. ECDSA together with SHA-2 or SHA-3 are possible using [13]. The scenario maps to the BCCSP interface commands Sign and Verify.
DLT#3 DLT#Encrypt/Decrypt	This scenario allows a Fabric node to encrypt or decrypt data whenever this operation is required. The scenario maps to the BCCSP interface commands Encrypt and Decrypt. NOTE As of November 2019, these commands are stubs within [13].
DLT#4 DLT#PrivateData	This scenario allows creation of private data by using cryptographic hashes and random number generation (salted hashing).
DLT#5 DLT#SecureCommunication	This scenario allows secure communication between peer and ordering nodes using TLS with server authentication or client (and server) authentication.
DLT#6 DLT#MSP	The Membership Service Provider scenario abstracts away all cryptographic mechanisms and protocols behind issuing certificates, validating certificates, and corresponding user authentication. Authentication uses DLT#2.

C.1.10 GTA management

C.1.10.1 Overview

The solution that implements the GTA-API, that is, the generic trust anchor (GTA) itself, provides security mechanisms that are essential to the device it is integrated and used in. It is required that GTA itself is manageable in a way that strictly preserves this security.

Subclause C.1.10 summarizes all scenarios which do not have application- or device-specific functionality in their focus, but instead the management of the GTA implementation itself. This scenario collects GTA specifics beyond the general Device Management scenario.

C.1.10.2 Scenario overview

Table C.12 summarizes all scenarios which focus on the secure GTA management.

Table C.12 – Scenarios for GTA management

Reference	Scenario description
GMG#1 GMG#GTA-Update	The asset owners want to update their current GTA implementation using GTA-API, for example, if migration to new cryptographic parameters or algorithms is required. This update is to be done in a way that the integrity of the GTA implementation is preserved during the whole update process. The latter includes that only authorized updates are allowed, and that downgrading is not possible. Update processes can be restricted to certain parts of the GTA implementation.

C.2 Security requirements for security scenarios

C.2.1 General

Clause C.2 gives an overview of the security requirements and the corresponding scenarios for which the security functionality is required.

Requirements that are given by Table C.13 hold for all scenarios and have been derived from a more general perspective.

All security requirements except those requirements given by Table C.13 have been directly derived from the different scenarios that are described in Clause C.1 and are listed in respective tables Table C.15 to Table C.25.

C.2.2 General or nonfunctional requirements

For the different scenario categories, some general or nonfunctional requirements can be defined which should be considered for all scenarios given. These requirements are denoted in Table C.13.

Table C.13 – General or nonfunctional requirements

General requirement	Description
Scalability	The IT security solution that is to be defined should be scalable so that it can be adapted to the requirements of various users/applications.
Cost efficiency	The security concept should consider economic considerations. These considerations include: cost for the implementation, cost for maintenance or time-to-market.
Algorithm agility	• Only state-of-the-art security measures are used. • Where possible, the security measures are updatable by a software/firmware update when they are no longer considered reliable.
Integration with existing libraries	API documentation provides guidance on how to integrate the API with existing security libraries to allow brown-field applications to benefit from the new API. The following list gives examples of such security libraries. • OPC Foundation reference stack [2] • OpenSSL [3], for example, reference service provider implementation using the GTA • mbedTLS [4]
Testability	It is possible to test API implementations for conformity with this document. This conformity test can result in some specific quality attestation.
Robustness against implementation attacks	API developers should take implementation attacks like fault attacks or side-channel analysis into account as far as possible. This requirement especially applies to the specific scenarios that are to be supported by their API implementation.