Deliverable D3.3

Integrated ARMOUR experimentation services with FIT IoT-LAB testbed

Version
Version 1.1

Lead Partner
Inria

Date
18/04/2017

Project Name
ARMOUR – Large-Scale Experiments of IoT Security Trust
Dissemination Level

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>PU</td>
</tr>
<tr>
<td></td>
<td>CO</td>
</tr>
<tr>
<td></td>
<td>CI</td>
</tr>
</tbody>
</table>

Abstract

D3.3 “Integrated ARMOUR experimentation services with FIT IoT-LAB testbed” describes the actual integration of the components and the developments made by adapting FIT IoT-LAB, FIT Cloud for providing relevant services for ARMOUR.

This document describes the principles underlying the presented integration with the FIT IoT-LAB testbed, and focuses on a new long-term general architecture for this integration. It describes the design principles of this architecture and their declination though extensions and adaptations of the components of ARMOUR that are developed in other workpackages. It includes the description of the underlying interconnection (service bus, protocols) between these components, and a description of the control and data planes (and related design principles). The document also describes the addition of FIT Cloud platform in the architecture for supplementary services. Furthermore, it provides a more detailed focus on a baseline, the common occurrence in many experiments (and in many IoT deployments), of the following scenario: an IoT device connecting to an external server through a gateway. We provide a detailed analysis of relevant design and implementation aspects of this scenario.

Disclaimer

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 688237, but this document only reflects the consortium’s view. The European Commission is not responsible for any use that may be made of the information it contains.
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>29/12/2016</td>
<td>General outline</td>
<td>Inria</td>
</tr>
<tr>
<td>0.2</td>
<td>22/01/2017</td>
<td>General guidelines of the integration</td>
<td>Inria</td>
</tr>
<tr>
<td>0.3</td>
<td>29/01/2017</td>
<td>Added FIT Cloud, integrated feedback for partners, detailed last section</td>
<td>UMPC, Inria, all</td>
</tr>
<tr>
<td>1.0</td>
<td>31/01/2017</td>
<td>Proof-reading and minor corrections</td>
<td>UMPC</td>
</tr>
<tr>
<td>1.1</td>
<td>18/04/2017</td>
<td>Revised version (after review)</td>
<td>Inria, UPMC</td>
</tr>
</tbody>
</table>
# Table of Contents

1 Executive summary .......................................................................................... 6

2 Introduction ...................................................................................................... 7

3 Integrating Components and FIT services ......................................................... 9
   3.1 Integration Overview .................................................................................. 9
   3.2 Integrated ARMOUR Components .......................................................... 11

4 Design: Principles of the Integration ................................................................ 13
   4.1 Integration through Service Bus ............................................................... 13
   4.2 Component and Control .......................................................................... 14
   4.3 Decoupling of Components ..................................................................... 15

5 Actual Integration ............................................................................................. 17
   5.1 Protocol for the Service Bus .................................................................... 17
      5.1.1 Service Bus Protocol ........................................................................ 17
      5.1.2 Security of the Service Bus ............................................................. 17
      5.1.3 Communication Patterns on Top of Publish/Subscribe .................... 17
   5.2 Integration of FIT IoT-LAB Services ....................................................... 18
   5.3 Integration of FIT Cloud Services ............................................................ 20
   5.4 Integration of Experiment Software ......................................................... 21
      5.4.1 Tested Software ................................................................................ 21
      5.4.2 Experiment Agent and Controller ..................................................... 22
   5.5 Integration of Testing Tools Software ....................................................... 22
      5.5.1 Upper Tester for IoT Software .......................................................... 22
      5.5.2 Integration of Testing Tools ............................................................... 22
   5.6 Inter-Component Communication ............................................................. 23

6 Details of the Integration in Typical IoT Scenario ............................................. 25
   6.1 Typical IoT Scenario ................................................................................ 25
   6.2 Mapping of a Typical IoT Scenario on FIT IoT-LAB and FIT Cloud Services ...... 25
6.3 Prior Proof of Concept of Test of a Typical IoT Scenario (EXP1) ......................26
6.4 Evolving the Proof of Concept towards Presented Architecture ...................30
6.5 Design and Implementation of Serial Agent .................................................31
  6.5.1 Overview: from FIT IoT-LAB to ARMOUR Service Bus ..........................31
  6.5.2 Serial Agent global topics .......................................................................34
  6.5.3 Node topics ..............................................................................................34
  6.5.4 Line redirection topics .............................................................................35
  6.5.5 Raw redirection topics .............................................................................36
7 Conclusion .......................................................................................................38
1 Executive summary

This document is related to WP3 and its task T3.2 (and more precisely task T3.2a): the main purpose of this document is to present the actual integration of the Security & Trust Experimentation Services/Framework (developed in WP2) with FIT IoT-LAB and the addition of the extended services such as FIT Cloud.

The work is based on the input from other packages: the deliverable D1.1 “ARMOUR Experiments and Requirements” defines the requirements of each experiment; the deliverable D1.2 “ARMOUR Experimentation approach and plans” describes how each of the experiments will be performed in the testbeds; the deliverable D2.2 “Test generation strategies for large-scale IoT security testing” includes a more detailed description of the best practices for testing IoT systems including the multiple test configurations; the deliverable D2.3 “Testing Framework for large-scale IoT Security Testing” includes a part devoted to test execution on FIT IoT-LAB; the deliverable D3.1 “Testbeds Detailed Analysis and Integration Guidelines” describes pre-existing services and features of testbeds, and provides guidelines for their use.

Within the ARMOUR project, the integration of the testbed is a crucial aspect of the test execution. In the initial phase of the project, analysis and guidelines have been provided, along with experiments requirements (see deliverables). This was the basis for following proofs of concept that glued all the components of ARMOUR security testing framework targeting one specific experiment, EXP1) in order to explore how the different components can be best integrated: by nature, the proof of concepts involved manual interaction with the components, along with specific (non-generic) software developments, and numerous aspects that were not sufficiently general to carry other experiments, scenarios, or tests.

This document is the outcome of the second phase of the project, analysing the lessons of the proofs of concepts, and capitalizing on progress in the definition and developments of the ARMOUR Security & Trust Experimentation framework: we provide an unified and general architecture that provides the features to allow for seamless integration of ARMOUR components, and for genericity. It will also allow reuse in the exploitation of ARMOUR. Care has been taken for the architecture to be sufficiently flexible so that it can gradually introduced, e.g. not requiring complete changes at once.

This document thus describes the general organization of this integration of FIT IoT-LAB and FIT Cloud in the rest of the ARMOUR architecture, and the converse integration of other components for execution within the testbed. We start from high-level principles, explore and depict how other components can be integrated, and then provide finer details: through the analysis of one common IoT scenario, and the analysis of its impact on all the relevant ARMOUR components. Finally, as a representative example, we provide a fine-grained description of one integrated component (Serial Agent) that has been updated to follow this architecture.
2 Introduction

This document describes the technical integration of the ARMOUR Security & Trust Experimentation framework with the FIT IoT-LAB testbed. It describes the services that are available at milestone MS14 where FIT IoT-LAB (and FIESTA) testbeds are ready for running ARMOUR Security & Trust and experiment designers can access and use all services needed to start the test design. This document is related to WP3, task T3.2 (more specifically task T3.2a). Its relative position in the ARMOUR project is represented in Figure 1.

![Figure 1 - Position of the document within the ARMOUR project (D3.3 and task 3.2a)](image)

It is provides a framework for the triple integration of the 1) tools of the ARMOUR testing framework, 2) the testbeds, and 3) the experiments. Notably, it provides one well defined way to perform orchestration among the different entities. The interface with FIESTA-IoT will be further defined in D3.4.

The document aims at defining the seamless technical integration of the ARMOUR Security & Trust Experimentation framework with the FIT IoT-LAB testbed. The objective of the services provided by FIT IoT-LAB are:

- Provide seamless interconnection with automated testing tools
- Aim at providing a general framework for all experiments
• Provide flexibility for addition of specific experiment needs

The integration aims further at satisfying the requirements of the experiments for FIT IoT-LAB:

• Map the requirements of the experiments to the testbed features
• Explore the various identified alternatives for missing features

We also provide and implement upgrades of the services of testbeds.

The realization of the objectives is provided through the document: there is an identified need for an architecture that is sufficiently generic to be reusable across experiments and beyond the end of ARMOUR project. In particular, a more high-level set of tools will allow decoupling ARMOUR Framework from the testbeds. This is useful for the long-term even if specific scripts and developments are could be sufficient for each specific experiment in the short term.

The document is the outcome of efforts on the workpackage WP3 for achieving the objectives: it results in an architecture that is designed starting from clear principles; and in a description and analysis of how to apply the architecture with the existing components; it also results in providing details through the perspective on one common IoT scenario, and finally describes one upgraded FIT IoT-LAB service.
3 Integrating Components and FIT services

3.1 Integration Overview

The Figure 2 represents the general place occupied by the FIT IoT-LAB services within the ARMOUR Security&Trust Experimentation framework. They act as an interface between the testing tools (where the abstract security models and tests have been defined), and the actual platforms (FIT IoT-LAB, FIESTA-IoT) where the actual devices, and associated monitoring/instrumentations tools will be performed.

In addition, there is a general need for a place outside the experimental platforms for running tools, in general on a local machine, or in the cloud: this can be provided by another of the platforms in FIT, FIT Cloud, as represented in Figure 3 and further described in section 5.3.

An example of test execution, and experimentation is given in deliverable D2.3 “Testing Framework for large-scale IoT Security Testing”, and is the basis for a more precise analysis of the critical points. Indeed, it is identified that the actual operation of a test on FIT IoT-LAB has two interrelated aspects:
• The coordination of the different components from different sources (e.g. the IoT-LAB monitoring services, the automated security testing tools, the software from the experiments, …), during the execution of one test.

• The control and temporal aspect: it is linked to the manner by which resources are reserved, configured, put in place, before the experiment is started, then by which they are managed during the experiment, and finally to the way by which they are released at the end of the experiment.

The aspect related to the coordination of different components is symbolized in Figure 4. Running a security test is materialized through an experiment that involves: the experiment itself, and software related to the experiment (e.g. tested code), the testing tools, and the testbed where the experiment is running. The coordination has been identified as a major technical point for this integration.

The control and temporal aspects are related to what is represented in Figure 5, and represent the typical workflow of a test run on IoT-LAB (from the perspective of FIT IoT-LAB).

The two aspects described before can be summarily considered as being the control plane and the data plane respectively. This is further described in the next section, after the following description of ARMOUR components.
3.2 Integrated ARMOUR Components

In this section, we briefly describe the different software modules and programs involved in the execution of an actual ARMOUR test on the testbeds. In this document, we denote them components.

ARMOUR integrates software components of different nature, and their list is summarized in Table 1 (not necessarily exhaustive). They come naturally in the final realization of a test, after several prior steps in the formal methodology of ARMOUR, and they include:

- Software from the testing tools that applies the formal ARMOUR methodology:
  - The abstract test models, from the model-based testing approach and software tools to model formally such tests.
  - Test generation software that will automatically create test information such as: test cases, data, oracle, and test code: this is CertifyIt. In ARMOUR, there is a specific focus on the TPlan and TTCN-3 abstract test suites. Note that, for instance, the abstract test suites need not be specific to particular implementations of the code (hence the same abstract test could be envisioned for checking different implementations).
  - Test implementation: it is derived from the abstract test suites and requires adaptation layers specifically for ARMOUR, for experiments, and for the framework described in this document. In ARMOUR, we use Eclipse Titan (with additional ARMOUR-specific software support) to automatically generate an executable test suite (materialized by full-fledged C++ source code).
  - Test execution: the actual test execution is performed on FIT IoT-LAB (and using FIESTA-IoT), by compiling and running the execution test suite generated by Titan.

- Software from the experiments themselves. This includes the tested IoT software itself, but also the tools on the servers and other support software, for instance:
  - IoT software running as embedded software on the IoT nodes (e.g. M3 nodes).
  - IoT software running on the gateways (e.g. Linux A8 nodes), such as IPv6/6LoWPAN gateways (e.g. tunslip).
  - Other software participating in the experiment such as servers (authentication authorities, software distribution servers, …).

- Software from the testbeds FIT IoT-LAB and FIESTA-IoT. They are for instance described in deliverable 3.1 “Testbeds Detailed Analysis and Integration Guidelines” and related deliverables. This includes:
  - Tools to manage experiments (reserve, start, stop, …)
  - Tools to interact with nodes (UART, reset, …)
  - Tools to monitor the nodes and the experiments (sniffer, energy consumption, …)
  - Tools to interact with the servers of the testbeds (FIESTA-IoT and IoT-LAB).
  - Support tools for the cloud: FIT Cloud tools.

ARMOUR includes also additional software components necessary for performing the integration of the different software modules, described in this document.
<table>
<thead>
<tr>
<th>Name of component</th>
<th>Role in ARMOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smartesting Certiflyt</td>
<td>Modelling of the abstract tests, and generation of abstract test suites.</td>
</tr>
<tr>
<td>Eclipse Titan (with support for ARMOUR)</td>
<td>Specialisation of an abstract test suite and generation of the (source) code able to run the specialized test.</td>
</tr>
<tr>
<td>Executable Test Suite, code generated by Titan (with adaptation/extensions for ARMOUR)</td>
<td>Code that will manage and perform the actual test.</td>
</tr>
<tr>
<td>Experiment software: embedded software</td>
<td>Software of a specific experiment that is additionally running on embedded nodes (typically one or several applications on top of Contiki/OpenWSN/RIOT).</td>
</tr>
<tr>
<td>Experiment software: server, tools, …</td>
<td>Software of a specific experiment that is running on the gateway, on the FIT IoT-LAB front-end server, or somewhere else in a cloud service.</td>
</tr>
<tr>
<td>FIT IoT-LAB software</td>
<td>Various tools and libraries for FIT IoT-LAB.</td>
</tr>
<tr>
<td>FIESTA-IoT software</td>
<td>Various tools and libraries for FIESTA-IoT.</td>
</tr>
<tr>
<td>FIT Cloud support software</td>
<td>Various support tools and support libraries for the use of FIT Cloud.</td>
</tr>
</tbody>
</table>

*Table 1 - ARMOUR components*
4 Design: Principles of the Integration

The integration of ARMOUR Components is based on the design principles and on the module organization described in the following subsections.

4.1 Integration through Service Bus

The ARMOUR Security & Trust Experimentation framework running in the FIT IoT-LAB testbed, with FIESTA-IoT and FIT Cloud, is organized as: services connected through a service bus. All components participating in an ARMOUR test, i.e. automated testing components, experiment/tested modules, FIT IoT-LAB (or FIESTA-IoT), …, have an agent, playing the role of an “adaptor” which makes them operate as a service and allows them to connect with other parties.

As a result, the undefined “coordination/orchestration” of Figure 4, is materialized in the idealized integrated ARMOUR architecture, represented in Figure 6, through two principles:

- A common “service bus” interconnecting all parts. This approach is common for distributed systems with potentially a large number of decoupled components. By having a unique rendezvous point (ideally on a single, well-known server), the interconnection is made more manageable.
- Having an agent component for each of the interconnected parties, that acts as adaptor. The agent is responsible for communicating with well-defined protocol with other components of the ARMOUR project.

As represented in Figure 7, the connection between the component and the agent associated with the component may take different forms: the agent might be integrated with the component itself (e.g. same process, hence achieved through a modification of the software component), the agent might be a different process, communicating either through a specific means (e.g. interprocess communication), or even through the service bus itself.
4.2 Component and Control

The idealized architectural model in Figure 6 is sufficient to have an architecture that operates successfully once it has been configured at set up.

However, it is not sufficient to fully automate the ARMOUR Trust & Security Framework on FIT IoT-LAB, as it is missing some aspects that are related to the control plane. For instance: starting other components, discovering which other components are present, what is their state (including failures), waiting for events from other components, etc.

The control plane architectural scheme that is represented in Figure 8 is proposed for the ARMOUR integrated IoT-LAB services.

![Diagram of control plane](image)

**Figure 8 – Control plane: Factory-Controller-Agent pattern**

It is intended as a general “design pattern” rather than a fixed formal. It involves:
• ARMOUR components that participate in one actual test execution, and their associated component agents that are connecting them to the ARMOUR service bus (e.g. as represented in Figure 6).

• Controller: an agent that is interacting directly with a component agent (from the control plane point of view). The controller:
  o Is able to control (query the state, configure, ...) the component through the component agent,
  o Has been given the responsibility or/and ownership of the component that it controls,
  o Exports itself a thin control API. This API is used as a building block for managing several agents.

• Factory: it is an optional component that is able to create (or reserve/allocate/retrieve) instances of the other components. Several variations are possible depending on the specific circumstances:
  o If not all information is available for starting a component, then a fully automated factory might not be the best solution.
  o A factory might be integrated with the controller itself: in other terms, the controller can be a factory itself for the agents that it controls.

In order to provide support for large-scale experiments (large in terms of the number of involved components), the controller re-exports a control API (such as query/answer). This allows for complex interaction between the different controllers, which might be required in some scenarios.

Note that the factory is “optional” for practical reasons: although it would be present in general case, the operation of starting/creating a component and its component agent requires more (meta-)information than just controlling it. Rather than designing complex (meta-)information exchanges across several agents, it might be more efficient to keep factory and controller separate, and have the factory just delegate control.

4.3 Decoupling of Components

The general architecture is deliberately designed to favour decoupling and modularity. Decoupling enables individual testing of each component, and partial integration. The idealized integration is represented in Figure 9. It is intended to be a guideline, rather than defining rigidly how one should organize the components of an actual test execution. It integrates the two separations discussed previously:

Horizontally, separation across semantic functionality of the related components:

- Testing tools
- Experiment software
- Testbed tools and software (FIT IoT-LAB, ...)

Vertically, separation across specific function for the integration:

- Base component (pre-existing to ARMOUR)
- Component agents
- Component controller (component control agents).

The control plane and the data plane have an overlap in the following place: the agents (in the middle part). One reason is that there are different levels of control plane (e.g. controlling the agents or controlling the components/software of the experiment itself). One design criterion for features that could be implemented either in the controller or in the associated agent itself is the following: once the experiment agents have been configured, they should be able to operate ideally without the controller.

![Figure 9 - Modular architecture of integrated ARMOUR components](image)

The horizontal communication is at the choice of each component. Additionally, as previously explained, for the communication on the left part (e.g. between “Testing tools” and “Testing tools agents”, etc.), the means of communication and protocols need to follow a common, standardized, approach, as they are internal.
5 Actual Integration

5.1 Protocol for the Service Bus

One key support element for the architecture is the service bus on which all components are "plugged". It is discussed in this section.

5.1.1 Service Bus Protocol

The underlying protocol that is used for the communication as a software bus is MQTT (v3.1.1), as specified in:


Although the protocol itself is not primarily designed for this type of usage, it can be easily adapted to construct a service bus protocol - especially in light of the current trend in Information Centric Networking, ICN, where it is shown that sophisticated naming and simple publish/subscribe features are enablers for complex communication patterns.

Useful features of MQTT include:

- Security (through TLS),
- Single server (MQTT broker),
- Simple concept of topics, with simple hierarchy,
- Communication through publish/subscribe.

Most notably, publish/subscribe allows for “streaming” data.

In practice, aside MQTT session control (connect/disconnect/…), our usage of MQTT is entirely summarized by the following commands:

- Publishing data to a topic (prior subscription to the topic is unnecessary).
  - Data is a sequence of bytes
  - Topic is an utf-8 string. It is interpreted as a hierarchical namespace (using ‘/’ as a delimiter as for Unix file names).
- Subscribing to a topic (with optional pattern matching based on hierarchy).
- Unsubscribing.

5.1.2 Security of the Service Bus

MQTT brokers (including the one used in the ARMOUR project) offer a first layer of security through TLS connections. In our architecture, the MQTT broker is potentially installed in a FIT Cloud server.

5.1.3 Communication Patterns on Top of Publish/Subscribe

One missing feature in MQTT is the concept of request/reply. This can be simply constructed as follows:

- An agent that is offering a service as request/reply subscribes to a well-known topic.
- Another agent can send a request to the other agent by:
  - Pre-subscribing to a reply topic
  - Sending the request on the request topic
And the first agent, receiving the request, will process it, and send it back to the initial agent on the reply topic.

There exist many ways to select the naming in this request/reply pattern. They are generally based on:

- Having a unique identifier for the request and/or for the requesting node
- Selecting a return “path” (topic), based on this identifier

The request/reply pattern can be abstracted as:

- REQUEST(<name>,<command>,<parameters>,<return-info>,<request-id>)
- REPLY(<name>,<command>,<return-info>,<request-id>,<result>)

5.2 Integration of FIT IoT-LAB Services

In this section, we describe the FIT IoT-LAB services for which agents can be provided for the ARMOUR project. Main services are summarized in the Figure 10.

We revisit the list given in deliverable D3.1, by indicating how the integration of the services is done.

The philosophy of the services is to provide total remote access to the reserved nodes without any requirement or constraint: any language or OS can be used to design, build, and compile firmware and applications.
The services include:

- Control of the nodes themselves:
  - An agent can export the interface of the total remote access (in terms of control), through a service bus agent:
    - Reflashing the nodes,
    - Resetting the nodes,
    - Stopping the nodes,
    - Etc.
  - Direct access to a debugger server on each node. Thus debugging can be performed remotely on the node (such as executing code step by step). It is totally equivalent of having the node on its own desk and debugging it through a JTAG cable;
  - An agent can export the debugger API for IoT-LAB nodes for which it is supported. In practice, however this would imply two parts: a control part to select which nodes are debugged etc., and an interface for sending commands, which might well be the “gdb” command line interface over service bus, or other levels of control (such as exporting the openocd interface).
- Access to the serial ports of all nodes for a real-time interaction, with optional aggregation. The user can collect and aggregate of some/all serials ports of his/hers reserved nodes:
  - The FIT IoT-LAB tool providing such feature is the serial_aggregator. An agent exports the serial ports of all the nodes, on the service bus. It provides several features such as:
    - Getting output of the nodes in a line based format,
    - Getting output of the nodes in a “raw” based format,
    - Managing proper subscription to input/ouput of selected subsets of nodes,
    - Sending data to the nodes.
  - The publish/subscribe nature of the service bus fits perfectly the “streamed” nature of input and output of the node, and can be used for node selection.
- End-to-End IPv6 connectivity. Each node can be visible from Internet with end-to-end IP connection using IPv6 and 6LoWPAN for example;
  - An agent can manage the various software tools offering this function (tunslip for Contiki, through event bus in OpenWSN, ethos for RIOT, …). However, note that it would mostly act as a controller or as a factory.
- An accurate power consumption monitoring of every node. The monitoring of all nodes is done on the side so that applications are unmodified. Our philosophy is that the application deployed is the final application and not an instrumented of it with modified code in order to do this specific monitoring;
  - In the current FIT IoT-LAB infrastructure, a set of tools exports the power consumption through files written on-the-fly in the IoT-LAB front-end. Because of the dependency on underlying OML libraries, and on NFS file server, the currently service would not be in immediate real-time, but with a delay in the order of magnitude of several dozen of seconds.
- Packet sniffer of each M3 node. Again, the sniffer is on the side and does not require any modification of the application;
The case of the packet sniffer is similar to the case of the access of the serial ports of the nodes. The difference is that it involves more control (e.g., channel selection), and management of FIT IoT-LAB profiles. Along channel selection, we can envision features such as selection of the format, etc.

- Strong software and usage support with a set of detailed tutorials on a wide range of topics, OS support (Contiki, FreeRTOS, TinyOS, and RIOT) including full protocol stacks and communication libraries such as OpenWSN providing open-source implementations of IoT protocol standards;
- For ARMOUR, relevant agents could be imaged for controlling well known software, such as Contiki, OpenWSN, and RIOT (including the protocol on the serial port), in several relevant scenarios.

5.3 Integration of FIT Cloud Services

In this section, we are describing FIT Cloud service as one complementary place that allows users to deploy the non-embedded and/or additional support software. In some cases, such a cloud service is potentially required in order to plan, deploy and execute an experiment and/or to collect any suitable informations and results. FIT Cloud services are generally creating space for “Experiment software” as defined in section 3.1 of this deliverable and are available for all partners of ARMOUR project.

FIT Cloud is providing users with on demand (via web interface) cloud server instances with full administrative access. All VM’s are interconnected with internal IPv4 network and each has a direct connection to IoT other services and devices via IPv6 link(s).
5.4 Integration of Experiment Software

5.4.1 Tested Software

As described previously, one can split experiment software in two categories: embedded software, and other (non-embedded) software.

One key limitation is that embedded software is constrained (in terms of RAM, flash, and even CPU). Therefore, any modification/instrumentation for testing purposes poses challenges, and it is proposed that in general:

- A minimal instrumentation is provided, through interaction on the serial port (for instance command line shell for Contiki/RIOT, through proper HDLC messages for OpenWSN, …)
- The bulk of the responsibility for adapting the internal communication protocol on the serial port and the protocols for interacting with other ARMOUR components can provided by one specific experiment agent, if necessary.

The suggested integration is represented in Figure 12. Because the serial port is managed by FIT IoT-LAB agents (serial aggregator) represented on the right of the figure, the communication goes through these agents. Then the experiment agent communicates to other ARMOUR components through the service bus, and notably with testing tools (see section below).

The experiment agents for embedded software will have commonalities between different experiments, either because shell commands are identical if the same embedded OS is used, or the behaviour is the same: “request”/“wait-for-a-predefined reply”. Because of this, code reuse is expected, and template agents are expected to be developed.

![Figure 12 – Integration of Instrumented/Tested Embedded Software](image)
5.4.2 Experiment Agent and Controller
A typical FIT IoT-LAB experiment (outside of the ARMOUR context) involves running one or several scripts for configuring, starting, managing, controlling different elements of the experiments. An ARMOUR experiment may adopt the same approach. However, in the general case, more complex interactions are expected, and notably for automating the experiment. Therefore, an agent designed to control the whole experiment can be introduced.

Similarly, to the embedded software agent, the experiment agent/controller is expected to have commonalities between the different experiments.

5.5 Integration of Testing Tools Software

5.5.1 Upper Tester for IoT Software
This section continues the discussion of the section 5.4.1, for integrating embedded software. The deliverable D2.2 “Test generation strategies for large-scale IoT security testing” specifies various strategies for testing IoT systems. The Figure 13 reproduces the general framework for testing a client node considered as a system-under-test. It involves the addition of an Upper Tester SUT (that should be closely linked to the client node), and an Upper Tester TS.

In our framework, the upper tester SUT can be separated in two parts: one part is the (lightweight) command module inside the SUT (embedded software) used for instrumentation. The other part can be implemented in a separated agent when the behaviour of the Upper Tester SUT is more complex and deserve a standalone agent. In that case, the link with “other ARMOUR component agents” in Figure 12 is the communication with an upper tester SUT agent (from the experiment agent for embedded software).

5.5.2 Integration of Testing Tools
The testing tools are integrated with the rest of the ARMOUR components: during the test execution, the “executable test suite” (with proper ARMOUR additions) generated by Titan, is what is actually running. Because the executable test suite generated by Titan is a standalone component, it might have some hard constraints (in the execution flow of the program for instance).
Then the same principle as for the embedded software can be adopted: have a minimal implementation inside the generated code of Titan and creating an external “testing tool” agent (potentially the Upper Tester TS of Figure 13) that is able to implement more complex logic or to manage more complex interactions, as represented in Figure 14.

![Figure 14 - Testing tools integration](image)

Because this relates exclusively to the communication of entities of the testing tools, any internal protocol can be used: HTTP+REST, COAP, MQTT, …

Another point is the separation of the logic between the “Upper Tester Test System” and “Upper Tester SUT” represented in Figure 13: they can be materialized by two different agents in the architecture; or their logic can be integrated into one common agent.

### 5.6 Inter-Component Communication

We reproduce the architecture of the Figure 9 with an emphasis of the most critical parts in the first steps of the project ARMOUR. It is represented in the Figure 15.

![Figure 15 - Critical inter-component communication](image)
The critical parts include communication between the different components through the software bus. This involves:

- Communication between testing tools and experiment agents
- Communication between testing tools and testbed agents
- Communication between experiment and testbed agents

For each of these communications, the precise format of the exchange can be defined independently.
6 Details of the Integration in Typical IoT Scenario

6.1 Typical IoT Scenario

As described in deliverable D2.2 (and using the same terminology), a common deployment in IoT consists of: 1) one server node (SN) providing advanced services, 2) one or several client node(s) that run(s) embedded software, 3) a middle node such as gateway/router. It is represented in Figure 16.

![Typical IoT communication scenario](image)

This scenario occurs independently of the ARMOUR framework. The next section describes how it can be instantiated in the ARMOUR framework.

6.2 Mapping of a Typical IoT Scenario on FIT IoT-LAB and FIT Cloud Services

When applying the methodology of the ARMOUR framework, the test execution will be materialized by mapping the different elements of the experiment (sensor, gateway, server) to elements in the experiment. The Figure 17 describes how the test can be mapped to the testbed services without the architecture described in this document. This is the same as what was actually implemented for the proof of concept in D2.2.

The elements are as follows:

- The tools, auxiliary support, and server parts of the experiments that are run in one server or several servers. FIT Cloud provides the perfect place to run such services, through virtual machines. They are:
  - The Titan testing tools and the generated executable test suite,
  - The server of the experiment,
  - The software that bridges the serial port protocol to actual IPv6 packets,
  - Optionally: the sniffer and packet analysis software (to attempt decrypt packets).
- The tools that are provided by FIT IoT-LAB. Some of them are available on the FIT IoT-LAB front-end at the site where the experiment is running (and can be tunnelled for instance through ssh):
The serial ports of the IoT-LAB M3 nodes of the experiments are available through one specific TCP port number
- Optionally: the radio sniffer is available on another specific TCP port number

**The gateway:**
- It is typically an experiment node of FIT IoT-LAB, e.g. an IoT-LAB M3 node, with special configuration to operate as a 6LoWPAN border router (BR), and also is radio-enabled with 6LoWPAN.

**The sensor:**
- It is also typically an experiment node of FIT IoT-LAB, e.g. an IoT-LAB M3 node, with application software (using for instance COAP+DTLS),

![Diagram of IoT-LAB network](image)

*Figure 17 - Test of the typical IoT scenario*

### 6.3 Prior Proof of Concept of Test of a Typical IoT Scenario (EXP1)

Most ARMOUR experiments encompass some variation of this scenario, including the EXP1, where the server represents an authentication authority server. The proof of concept is further described in deliverable D2.3 for instance, and we reproduce the steps here for analysis.

Overall, the objective of EXP1 is to secure the communication of one sensor by using encrypted channels and also encrypting their messages. The scenario is depicted in Figure 16.

In our case, the sensor requests a key by providing its specific attributes first. It does so through one CoAP+DTLS POST message. This message is forwarded by the gateway to the server, that is called the Attribute Authority in this scenario. The Attribute Authority
processes the attributes of the request, in order to see if it is well formed and if access can be granted. If validation succeeds, the Attribute Authority answers with a CoAP message that includes the already generated key (called a CP-ABE key in the scenario).

In the latest iteration of the proof of concept experiment, these steps were followed:

- Prior software adaptation, development and compilation of software,
- Experiment reservation, and initial configuration of FIT IoT-LAB,
- Configuration of additional elements or tools, including experiment-specific elements,
- Test execution: running the test as an “IoT-LAB experiment”, coordinating with testing tools (TITAN test execution), and providing the testbed tools based on FIT IoT-LAB features.

For the experiment reservation, one reserves two M3 nodes on the FIT IoT-LAB platform. The detailed steps followed are:

1. First log into the FIT IoT-LAB portal (https://www.iot-lab.info)
   - Click on "New Experiment"
   - Set a name and duration (we recommend at least 20 minutes to have time for the rest of the steps)
   - Selection of nodes: order them by type and select m3:at86rf23. Select a number of nodes=2
   - Click on "Next" and "Submit"

Our platform allows the modification of the configuration after the reservation: one example if for changing the software running on the IoT-LAB nodes. In experiment 1, the firmware updates on the nodes can be performed with the following commands:

```bash
auth-cli -u user
node-cli --update border-router.iotlab-m3 -l grenoble,m3,8
node-cli --update control-manager.iotlab-m3 -l grenoble,m3,9
```

Then we establish connectivity with them (tunnelling, forwarding, address configuration, etc.), by having tools them ready and running on the proper server: in the latest iteration of the test of EXP1, this was done in one the ARMOUR VMs of FIT Cloud.

One of the generated test cases was executed on FIT IoT-LAB in the presented scenario and with the following specificities:

- The sensor is the SUT,
- The Titan testing tool (e.g. executable test suite) represents the Attribute Authority (AA). In essence, the logic of the AA (along with request analysis and reply generation) was re-implemented in Titan.
The Titan testing tool and its generated code are on the same FIT Cloud virtual machine as the sniffer and the gateway tunnel (it is tunslip, providing serial to IPv6 conversion). The whole experiment was launched with Titan as an AA in the virtual machine, while the gateway and sensor where in FIT IoT-LAB.

The test verified the successful exchange of information between both the AA and the sensor.

The Figure 18 illustrates the execution of this test, and also proves a first level of integration and communication between the various modules of the first version of the Large-Scale MBT Security testing framework.

```
TITAN, as it is shown in this figure, is able to reply the node of FIT IoT-LAB with the corresponding message. This is only the first exchange of messages that is done in order to send the CP-ABE key to the sensor.

This experiment has shown that a smoother integration needs to be performed between TITAN and other entities, because of issues in the mechanism used for transmitting blocks of information, and because there was a mismatch in the COAP implementation and COAP option use.
```
Figure 19 - Test of the typical IoT scenario with proposed architecture
6.4 Evolving the Proof of Concept towards Presented Architecture

The Figure 19 represents one possibility for performing the same test of EXP1 with the proposed architecture. Several ways are possible for mapping the test experiment on the architecture, but this is one of the most general one (in practice fewer agents would be present, because some of the control is not needed).

The some of the advantages of the architecture in Figure 19 can be identified on this example:

- **Coordination.** One first advantage is obvious when considering what happened in the proof of concept in Figure 18. Indeed, in the proof of concept, although the components were run from one unique VM (now in FIT Cloud), they were all launched manually. The base idea of interconnecting the components through additional agents that are interconnected through a service bus is what will allow automation of this part, through coordination. The upper testers (agents), the serial agent, and the sensor agent are the main parties involved in the coordination of this test.

- **Minimal instrumentation of the tested software.** This is addressed in the work packages WP1 and WP2, but we want to emphasize that the architecture supports well the following idea: testing software with minimal modifications for instrumentation. Indeed, in the proof of concept, the logic of the AA had to be re-implemented, along with a different implementation of the protocol (COAP), which caused: additional burden in development time (AA logic) and maintenance (adjusting COAP implementations to operate in identical ways both in actual AA, and in AA emulation). The minimal instrumentation is possible through the upper tester associated to a lightly instrumented software (sensor, server), and that connects indirectly through successive layers which are responsible for translating operations:
  - Sensor agent, which is able to translate commands to the nodes, and also parse the result and send it back to the upper tester.
  - Serial agent, which provides a service bus interface to the nodes input/output.

- **Experiment configuration:** the “experiment space”, e.g. the various agents related to the experiment, is one of the places were the proper information, data, configuration, embedded software is available. Experiment-related agent in the control plane will thus be able to configure automatically the firmware update of the nodes (border router, control manager codes), when they are implemented. They will rely on some “node control agent” providing a service bus interface to the relevant FIT IoT-LAB functionalities.

- **Further tool configuration** (tunnelling, forwarding, address configuration, etc.) will be performed by:
  - The FIT IoT-LAB agents (and optionally if the case arose, the FIT Cloud agents) for the part related to the testbed such as: tunslip configuration.
  - Experiment control plane agents.

- **In an advanced version of the implemented architecture:** the part related to the manual reservation can be automated: this involves agents in the control plane, and factories. Indeed, this requires an identification of the agent that is the central point of responsibility for running the experiment, then establishing the proper control
channels through control plane and controllers, and retrieving and pushing relevant information to where it is required.

6.5 Design and Implementation of Serial Agent

6.5.1 Overview: from FIT IoT-LAB to ARMOUR Service Bus

One of the developments for the presented architecture is the support software for FIT IoT-LAB. In order to provide a better view of how the architecture operates, we specifically describe how the functionalities of the command line software serial_aggregator have been transformed into agents for ARMOUR Testbed service bus.

Figure 20 - Repository for IoT-LAB service bus software
For the project ARMOUR, developments related to the FIT IoT-LAB testbed support (and more) are maintained in a software repository (private github repositories for instance). This is where various ports of software (for instance Contiki, DTLS software, EXP1 software, FIT IoT-LAB scripts, …) are available.

The “serial agent” has it has been called previously in this document is available as a “Serial Redirection Agent”. It is located in the ARMOUR repository containing the software related to the agents for IoT-LAB (see also Figure 20), expending and adapting the IoT-LAB features to the service bus.

The service provided by the software `serial_aggregator`, is: to read/write the serial port of all/some nodes in the experiment (through internal IoT-LAB mechanisms), and to export it on standard input/output. More precisely:

- It is a tool that allows aggregating data results from many nodes at a time. It connects to several TCP connections and handles the received data.
- In the basic use, it aggregates all the serial links of an experiment and prints them to stdout.
- In addition, it parses standard input to allow the forwarding of messages to the nodes. It reads using 'readline' and offers a shell-like interface to the end-user. It has a specific syntax:
  - `' ' -> does not send anything to anyone, allows 'blanking' lines`
  - `' ' -> sends `' \n' to all nodes`
  - `'msg' -> sends 'msg\n' to all nodes`
  - `'m3,1-3+5;message' -> sends 'message\n' to nodes 'm3-[1, 2, 3, 5]'`
  - `'m3-1;message' -> sends 'message\n' to nodes 'm3-1'`
  - `'csv;message;with;semicolons' -> sends 'csv;message;with;semicolons' to all nodes as 'csv;' is not a valid node identifier`

As shown, the implementation of this offered service is closely linked to the philosophy of the command line and the use of stdin/stdout (as several other FIT IoT-LAB tools). This is perfect for writing simple or sophisticated scripts, especially when direct observation is made between each run of a tool.

For full automation, for the reasons previously described, the service-bus architecture is more convenient, and hence modifications were performed. This was done through the following steps:

- The existing functionalities have been analysed, and they have been expanded to for support of ARMOUR Security&Testing Framework.
  - This included control and configuration of the serial redirection functionality.
- Then mapping of the functionalities to the service bus has been designed:
  - Naming,
  - Definition of request/reply API,
  - Management of streamed data (serial data, and errors).

The service of the Serial Redirection Agent is still to provide access to the nodes serial link. However now:
• It allows for data streamed either as “line-separated” or “raw” data (instead of just “line-separated” data),
• It allows for dynamic on-the-fly subscription to the serial port of specified nodes and unsubscribing,
• It has a mechanism (service bus topic) for reporting errors

For the naming, that is, the selection of the “topics” (in the service bus), the following principles are followed:

• By default, all ARMOUR services connected on the service bus, should take as configurable parameter one prefix denoted “{prefix}” that will be prefixing all the topics used.
• One strong choice was to adopt the philosophy that an agent should receive and write information only within a well-defined “namespace” that is therefore reserved to it. For the serial redirection agent, the namespace is: “{prefix}/iot-lab/serial/{site}”, where “{site}” represents the name of the IoT-LAB site.
• A certain number of global commands and the general error topic are directly one subtopic of the serial redirection agent namespace:
  o “stopall” (command), “error” (topic)
• All the other commands operate on individual nodes. Nodes have well defined names within IoT-LAB (such as “m3-4”, for the 4th node in the platform that is an M3 node), and they are used for the naming as “{archi}/{num}” where the components are respectively the architecture of the node (m3, a8, …) and the index of the node (1,2,3 …).
  o The commands are: “start”, “stop”
  o Once started, the data can be read and written through a subtopics of “data”: it can be read from “out” and written to “in”
  o Additionally, the output is available either in line mode (delimited by line terminators), or in raw mode (no terminators): the functionalities are available through different namespaces, where respectively “line” and “raw” appear in the topic name.

The exact topic choices are summarized here:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial agent</td>
<td></td>
</tr>
<tr>
<td>{prefix}/iot-lab/serial/{site}</td>
<td></td>
</tr>
<tr>
<td>{serialagenttopic}/error</td>
<td>Error</td>
</tr>
<tr>
<td>{serialagenttopic}/stopall</td>
<td>Request</td>
</tr>
<tr>
<td>Node</td>
<td></td>
</tr>
<tr>
<td>{serialagenttopic}/{archi}/{num}/ctl/stop</td>
<td>Request</td>
</tr>
</tbody>
</table>
The details of the provided features are described in the next section with an emphasis on their naming (topics).

### 6.5.2 Serial Agent global topics

#### 6.5.2.1 Error Topic
Asynchronous error messages are posted to the “error” topic. Failures due to request format are not duplicated here.

- `{serialagenttopic}/error`

#### 6.5.2.2 Stop all redirections
Stop all started serial redirections.

**stopall request:**

<table>
<thead>
<tr>
<th>Topic:</th>
<th><code>{serialagenttopic}/stopall</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message</strong></td>
<td>Topic</td>
</tr>
<tr>
<td><strong>Request</strong></td>
<td><code>{topic}/request/{clientid}/{requestid}</code></td>
</tr>
<tr>
<td><strong>Reply</strong></td>
<td><code>{topic}/reply/{clientid}/{requestid}</code></td>
</tr>
</tbody>
</table>

### 6.5.3 Node topics

#### 6.5.3.1 Stop redirection
Stop the serial redirection of one given node (no matter what the current mode is).
6.5.4 Line redirection topics

These are topics to access to the node serial port redirection in ‘line’ mode. The redirection must first be started in line mode to have line-formated output.

Data received from the node are split at “newline” characters (with newlines stripped).

As “newline” characters are not used in multi-bytes utf-8 characters, each line remains a valid utf-8 string if it was one initially. Invalid utf-8 strings sent by the node will stay invalid.

6.5.4.1 Start redirection in line mode

This starts the serial redirection of one node in line mode.

line/start request:

<table>
<thead>
<tr>
<th>Topic:</th>
<th>{serialagenttopic}/archi/num/line/ctl/start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Topic</td>
<td>Topic</td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>Request</td>
<td>{topic}/request/{clientid}/{requestid}</td>
</tr>
<tr>
<td>Payload</td>
<td>empty</td>
</tr>
<tr>
<td>Reply</td>
<td>{topic}/reply/{clientid}/{requestid}</td>
</tr>
<tr>
<td>Payload</td>
<td>empty or error_msg</td>
</tr>
</tbody>
</table>

6.5.4.2 Stop line redirection

Equivalent to “Stop redirection”, and is added here for completeness.

line/stop request:

<table>
<thead>
<tr>
<th>Topic:</th>
<th>{serialagenttopic}/archi/num/line/ctl/stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Topic</td>
<td>Topic</td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>Request</td>
<td>{topic}/request/{clientid}/{requestid}</td>
</tr>
<tr>
<td>Payload</td>
<td>empty</td>
</tr>
<tr>
<td>Reply</td>
<td>{topic}/reply/{clientid}/{requestid}</td>
</tr>
<tr>
<td>Payload</td>
<td>empty or error_msg</td>
</tr>
</tbody>
</table>
6.5.4.3 Line redirection
This enables serial redirection of text lines with newline characters removed (\n).

One message is received per line, and when sending a message, the newline character is automatically added.

<table>
<thead>
<tr>
<th>Text line serial redirection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topic:</strong> {serialagenttopic}/{archi}/{num}/line</td>
</tr>
<tr>
<td><strong>Message</strong></td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td><strong>Input</strong></td>
</tr>
</tbody>
</table>

6.5.5 Raw redirection topics

6.5.5.1 Start redirection in raw mode
This starts serial redirection of one node in raw mode.

<table>
<thead>
<tr>
<th>raw/start request:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topic:</strong> {serialagenttopic}/{archi}/{num}/raw/ctl/start</td>
</tr>
<tr>
<td><strong>Message</strong></td>
</tr>
<tr>
<td><strong>Request</strong></td>
</tr>
<tr>
<td><strong>Reply</strong></td>
</tr>
</tbody>
</table>

6.5.5.2 Stop raw redirection
It is equivalent to “Stop redirection” and is added here for completeness.

<table>
<thead>
<tr>
<th>raw/stop request:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topic:</strong> {serialagenttopic}/{archi}/{num}/raw/ctl/stop</td>
</tr>
<tr>
<td><strong>Message</strong></td>
</tr>
<tr>
<td><strong>Request</strong></td>
</tr>
<tr>
<td><strong>Reply</strong></td>
</tr>
</tbody>
</table>
6.5.5.3 **RAW redirection**
This is serial redirection for handling raw data.

Group of bytes are sent when received from the node redirection. No specific transformation is done when receiving or sending.

<table>
<thead>
<tr>
<th>RAW serial redirection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topic:</strong></td>
</tr>
<tr>
<td><strong>Message</strong></td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td><strong>Input</strong></td>
</tr>
</tbody>
</table>
7 Conclusion

The efforts of the task 3.2a of the WP3 have started from the detailed analysis of the target testbeds for IoT Security & Trust framework, from the prior defined guidelines, and from other outcomes from preceding work packages. This allowed us to define a new architecture for the integration of the components of the ARMOUR Large-Scale Testing Framework. Among other material, it used the output of the WP1 and WP2. One of the outcomes of this task is the present document D3.3 “Integrated ARMOUR experimentation services with FIT IoT-LAB testbed”. It will be complemented later by the document D3.4 “Integrated ARMOUR experimentation data and benchmarks with FIESTA testbeds”.

In the present document, we extensively described an advanced architecture, its principles, and its integration with the ARMOUR components on the target testbeds of the ARMOUR project: FIT IoT-LAB. It is the outcome of the second phase of the project, analysing the lessons of the proofs of concepts, and capitalizing on progress in the definition and developments of the ARMOUR Security & Trust Experimentation frameworks: we provide an unified an general architecture that will provide the features to allow for seamless integration of ARMOUR components, and for genericity (reuse in the exploitation of ARMOUR). Care has been taken for the architecture to be sufficiently flexible so that it can gradually introduced, e.g. not requiring complete changes at once.

This document thus describes the general organization of the integration of FIT IoT-LAB (and FIT Cloud) in the rest of the ARMOUR architecture, and the converse integration of other components for execution within the testbed. We start from high-level principles, and explore and depict how other components can be integrated, and finally provide finer details: through analysis one common IoT scenario, and through the analysis of one integrated component with its impact on all the relevant other ARMOUR components. We conclude with a representative example: a fine-grained analysis and description of one FIT IoT-LAB service (Serial Agent) that has been updated to follow this architecture.