X-by-Construction Design Framework for Engineering Autonomous and Distributed Real-time Embedded Software Systems

Research and Innovation Actions

Horizon 2020, Topic ICT-50-2020:
Software Technologies

Grant agreement ID: 957210

– Deliverable –

D4.1: Software System Specification for Trustworthy and Secure Computing Platforms

---

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957210.

Document information

Document title:	Software System Specification for Trustworthy and Secure Computing Platforms
Work package:	WP4
Editor:	UOP
Author(s):	UOP, KIT, VECTOR, QUB, FENTISS
Reviewer(s):	KIT, DLR
Document type:	Report
Version:	1.1
Status:	Released
Dissemination level:	Public

XANDAR consortium

No.	Short name	Name	Country
1	KIT	Karlsruher Institut für Technologie	Germany
2	UOP	University of Peloponnese	Greece
3	DLR	Deutsches Zentrum für Luft- und Raumfahrt	Germany
4	AVN	AVN Innovative Technology Solutions Limited	Cyprus
5	VECTOR	Vector Informatik GmbH	Germany
6	BMW	Bayerische Motoren Werke Aktiengesellschaft	Germany
7	QUB	The Queen's University of Belfast	United Kingdom
8	FENTISS	Fent Innovative Software Solutions SL	Spain

Copyright & disclaimer

Document revision history

Version	Date	Comments
1.0	2021-06-29	Publication of the final version.
1.1	2021-08-04	Correction of erroneous references to figures.

About this document

The document describes the activities performed in Task 4.1 of the XANDAR project. The goal of Task 4.1 is to specify the main components of the XANDAR software platform and the respective run-time system that will enable the online monitoring, self-adaptation and self-healing mechanisms of the projects. The first part of this deliverable describes the two main components of the XANDAR run-time engine, the XtratuM hypervisor of FENTISS and the AUTOSAR adaptive API, and their adoption to the XANDAR platform.

The first part of this deliverable is to describe the XtratuM hypervisor from FentISS, since it is the main component of the run-time engine of XANDAR.

The second part of this deliverable presents the HW and SW monitors that will be automatically generated by the XANDAR toolchain as well as the integration of the monitors in the XtratuM hypervisor.

The third main part of this deliverable is devoted to present the initial plans for three important parts of the XANDAR toolchain: the code transformations for reliability enhancements, our analysis for source-to-source code transformation frameworks, and the security extensions of the XANDAR platform. The following part presents the XANDAR support for ML/AI applications. Finally, we also describe the platform interfaces to the V&V framework of the project (implemented in WP5).

Table of contents

List of figures

Glossary of terms

AA	Adaptive Application
AP	Adaptive Platform
AST	Abstract Syntax Tree
AI	Artificial Intelligence
API	Application Programming Interface
CFT	Cache Fault-Tolerance
ECU	Electronic Control Unit
EM	Execution Management
GPU	Graphics Processing Unit
FDIR	Failure Detection, Isolation and Recovery
FP	Floating Point
HM	Health Monitor
INT	Integer
IPC	Inter-Process-Communication
LSM	Local Security Monitor
IMA	Modular Avionics architectures
MMU	Memory Management Unit
OS	Operating System
PVEE	PVEE
PHM	Platform Health Management
RTE	Run-Time Environment
RSM	Runtime Security Manager
ODD	Operational Design Domain
SWC	Software Component
TSP	Time and Space isolation
WCET	Worst-Case Execution Time
XNG	XtratuM Next Generation

1 Introduction

The XANDAR platform mainly consists of multicore SoCs that contain processing elements, memory elements, I/O elements, interconnect elements, and other hardware elements. The processing elements may refer to either single-core, multi-core, or special processing units. Special processing units are restricted to memory-mapped accelerators. In the context of XANDAR, they specifically refer to GPU or AI accelerators. The memory consists of caches, shared main memory, and local scratchpad memory. I/O refers to communication I/O interfaces that allow data exchange with any kind of any connected device, such as sensors, other controllers, or a cloud infrastructure. A hardware platform also consists of additional hardware elements such as Memory Management Units (MMU) for enabling safety and security features.

XANDAR will target high-performance multi-core SoCs such as the Renesas RCAR or NVIDIA Xavier series. These provide the necessary processing power to handle the rising level of computationally intensive software tasks such as AI. At the same time, XANDAR strives to support low-level controllers such as the AURIX. The latter is typically used for realizing functionality with hard real-time constraints. These may impose additional restrictions on the basic machine model, especially in regards to predictability, allowing WCET analysis tools to calculate clear upper bounds. The following figure depicts an abstract diagram of the XANDAR software architecture.

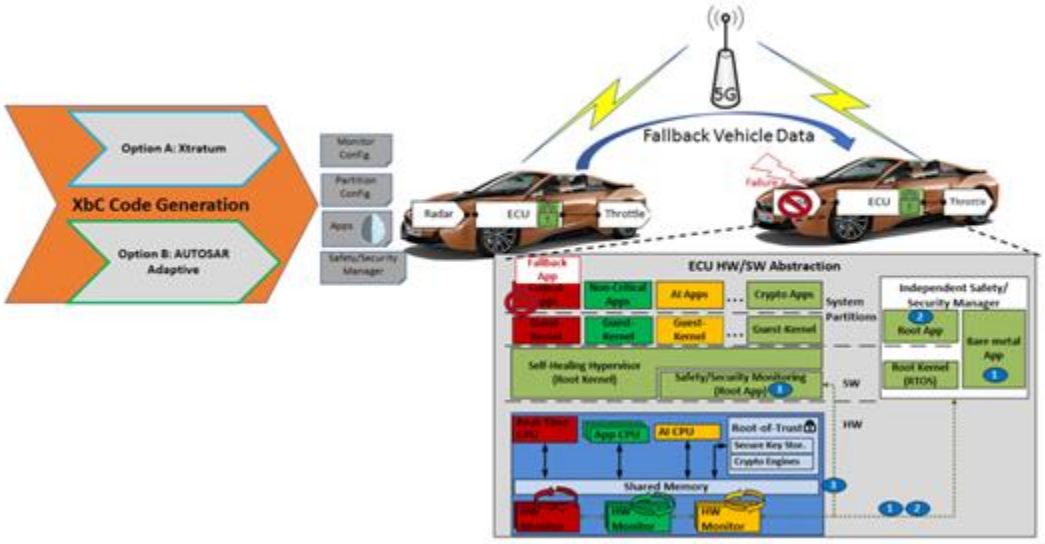

Figure 1-1: The XANDAR run-time layered architecture

The XANDAR run-time system consists of two integral parts: XtratuM hypervisor and the AUTOSAR adaptive platform. Both parts are described in this deliverable. On top of this, this deliverable contains information about the HW and SW monitors (automatically generated by the XANDAR toolchain), the proposed code transformations for reliability enhancements and our analysis for a source-to-source code transformation framework, the security extensions of the XANDAR platform and the support for ML/AI applications. Finally, we also describe the platform interfaces to the V&V framework of the project (implemented in WP5).

2 The XANDAR run-time system

The two main components of the XANDAR run-time system are the XtratuM Next Generation (XNG) hypervisor provided by the partner FENTISS and the AUTOSAR adaptive platform.

In the remainder of this section we will describe the main features of both components as well as the proposed extensions required to support the XANDAR toolchain and use-cases.

2.1 An overview of the XNG hypervisor

The XNG hypervisor is a software layer enabling the execution of several software systems on a hardware host machine. Each application is executed in an isolated virtual environment, that is, a partition, that mimics the behaviour of the underlying hardware architecture. The XNG hypervisor is aimed at real-time/dependable systems implementing critical functions. The configuration of the hypervisor is performed statically during the system design phase.

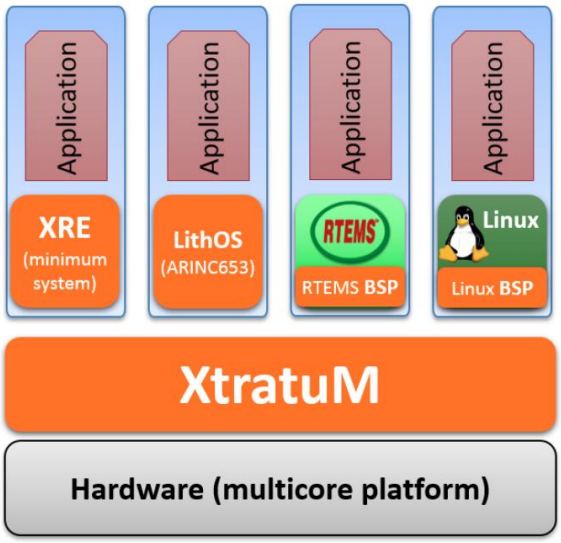

Figure 2-1: XNG overview

The XNG hypervisor executes with the maximum processor privilege level to access all the system resources and to enforce time and space isolation (TSP). A partition is an abstraction isolating a software application in time and space.

A hypervisor partition can be provided with different execution environments as follows.

- **XRE**: Minimal “C” run-time environment. It provides a virtual exception table, processor registers initialization and stack initialization. The entry point can be accessed by jumping to PartitionMain() “C” function. There are “C” handlers available for exceptions and IRQ. XRE does not provide concurrency thus no partition threads can be created.
- **LithOS**: Real-time execution environment oriented to Integrated Modular Avionics architectures (IMA). It provides an ARINC 653-P1 standard API and a partial implementation of the ARINC 653-P2 API. LithOS always runs as XNG guest OS (i.e. no stand-alone implementation).
  The services provided by LithOS, following the ARINC-653 standard, are related to:
  - Partitions, process and timing management

    - Communication between partition
    - Communication intra partition
    - Health monitoring
    - Extended services: Multiplan
- **Third-party guest OS**: Modified kernel to enable execution of a third-party operating system guest OS over XNG (modification is usually provided as a patch to the third-party kernel). This is the procedure used to integrate applications running over third-party operating systems like RTEMS and Linux in an XNG system.

2.1.1 Resource segregation

XNG guarantees time and space isolation to partitions. The performance and the behaviour of one partition cannot be affected by other partitions. The hypervisor is executed in privileged mode exclusively, unlike the partitions which run in user mode.

The virtualization method followed by XNG depends on the virtualization capabilities provided by the hardware. If the processor does not implement hardware-assisted virtualization then paravirtualization is used. The system is partially virtualized (paravirtualization) with a close performance to real hardware, the paravirtualization helps to simplify the virtualization implementation. The conflicting instructions are replaced by hypervisor function calls. The services are provided with the *hypercalls* but the native hardware access is not available.

2.1.1.1 Temporal partitioning

The vCPUs provided by XNG are assigned to partitions and scheduled according to a time-based activation schedule (cyclic scheduling) policy. A set of schedules can be defined in the configuration, where one is the active schedule. Through a Hypervisor service the active schedule can be changed during execution.

Each schedule is composed by:

- A Major Frame (MAF) defining the period of the schedule.
- A set of non-overlapped partition windows grouped by CPU in charge of running them.

Additionally, XNG allows the schedule to synchronise the start of every cycle with the arrival of an external signal corresponding to a configurable IRQ. Note that each partition in turn can implement an internal thread scheduling policy during its corresponding partition slots.

2.1.1.2 Spatial partitioning

Making use of the hardware components such as MMU or MPU, the hypervisor provides the memory access to partitions, thereby the partitions only can access to the preassigned memory areas. Any attempt to access a memory location not included in the defined area is detected and blocked by the hypervisor.

2.1.1.3 Hardware I/O access

XNG allows the system integrator to delegate a hardware platform I/O device to a partition. The access to IRQs and I/O ports (memory registers) of the device is configured in the configuration file.

2.1.2 Run-time policies and adaptation techniques

2.1.2.1 Time management

XNG provides partitions with a virtual representation of the underlying hardware. This environment is known as the PVEE (Partition Virtual Execution Environment). In the frame of this environment, each partition can use the resources below.

- Virtual system clock (vSysClock): This clock allows a partition to get the current system time (in µs) by using the XGetSystemTime() service.
- Virtual system timer (vSysTimer): This timer allows a partition to arm a virtual system timer (in µs) using the virtual system clock as base by using the XSetSystemTimer() service.
- Virtual execution clock (vExecClock): This clock allows a partition to get the time that the current partition has run (in µs) by using the XGetExecutionTime() service.
- Virtual execution (vExecTimer): This timer allows a partition to arm a virtual execution timer (in µs) using the virtual execution clock as base by using the XSetExecutionTimer() service.

Both the virtual system timer and the virtual execution timer, accept two arming-modes:

- **One-shot mode:** The virtual system/execution timer gets disarmed once its count expires.
- **Periodic mode:** The virtual system/execution timer is re-armed when its count expires.

2.1.2.2 Health monitoring

Inspired by the ARINC-653 XNG defines a HM service. The goal of the HM is to detect, react to and report hardware, partition and XNG failures. Therefore, it attempts to discover the errors at an early stage, trying to solve or confine the faulty subsystem to avoid a failure or to reduce its potentially harmful effects, preventing the propagation. The HM allows the system integrator to define a FDIR policy specific for the system and foreach partition event. This configuration is defined in the configuration file.

2.1.3 Safety-critical features

The temporal and spatial isolation of the XNG enforces the isolation of faults. This property is key for the implementation of highly safety-critical systems. Each partition can have different criticality levels without interfering with each other.

The fault management strategy of the hypervisor is established in two levels:

- The HM facility centralises the management of faults detected by all the components of the system. The HM is not itself responsible for the detection of the faults but instead centralises the reaction to and logging of the discovered faults.
- Additionally, the hypervisor has been designed to transit to a safe halt state when it is not possible to use the HM facility to handle the fault.

Faults detected by different components are reported to the HM that applies corrective measures to try to confine and correct the fault before it evolves into a system failure.

For example, in case of a partition or the hypervisor itself attempts to access a memory address outside of their allocated memory areas, the access is detected by the hypervisor (as a prefetch abort or data abort exception) and reported to the HM facility that may have been configured to halt the faulting partition or to reset the hypervisor.

The operation of the HM involves in the following elements:

1. Faults are reported to the HM facility as HM events. Regardless of which is the component that reports the fault, each event has an associated faulting element: either the hypervisor or any of the configured partitions.
2. When the faulting element is a partition, the HM applies a mapping function to convert from architecture-specific events to generic events.
3. The HM logs the event and the associated information in the corresponding log buffer.
4. The HM executes an action configured in the XCF. The hypervisor can be configured to execute a specific action for each hypervisor HM event. Specific actions can also be configured for each event and every partition.
5. Partitions access the HM logs to process the collected information according to the system needs.

The events supported by the HM are grouped according the following categories:

- Hypervisor events
  - **Hypervisor-generic events** are reported by the hypervisor itself when certain execution conditions are detected, possibly due to internal sanity checks (xHmInternalError) or due to robustness checks (xHmScheduleExtSynError).
  - **Architecture-specific events** indicate a fault in the hypervisor that has been detected by the notification of a processor exception.
- Partition events
  - **Partition-generic events** are generic events that are not directly raised by the hypervisor, but they are raised by the partition by means of the XRaiseHmEvent() service. Therefore, they do not have any semantics as far as the hypervisor is concerned, but their semantics is defined by one of their origins described previously. Additionally, the XReportHmEvent() service allows a partition to report a HM event, which will be reported in the HM log.
  - **Architecture-specific events** indicate a fault in the partition that has been detected by the notification of a processor exception.

    While the current version of the hypervisor does allow the use of architecture specific HM events as parameters to the XRaiseHmEvent() and XReportHmEvent() services, this use is discouraged since it breaks the portability of the partition. This behaviour may change in future versions of the software.

The actions can be classified according the following categories:

- Actions changing the execution state of the partitions, as for example xHmColdResetPartition. They only affect the partition for which the HM event is reported.
- Actions changing the execution state of the hypervisor, as for example xHmResetHypervisor. Note that these actions affect the complete system.

2.1.4 Security-related features

XNG executes with the maximum processor privilege level to have access to all the system resources, and is in charge of supervising the execution of the partitions running on the processor unprivileged mode. The hypervisor virtualizes the execution of privileged instructions, the access to privileged registers, the processor execution modes and the exception management.

2.1.5 XNG configuration

The configuration tasks involve the roles of System Integrator and the Partition Developers. They have to agree on the allocation of resources to the hypervisor and partitions during the design of the system, in order to get the XML-formatted XCF files. When the hypervisor starts its execution, during the system development phase or during the operation phase, it expects to have access to the configuration values in binary format.

To close the gap between the two steps, the XML-formatted XCF files (represented as XCF.xml) can be translated to a C counter-part (XCF.c) that can be translated to C code with the xcparser tool and then to the final binary (XCF.bin) using the GCC compiler. The complete process is depicted below. The tool Xoncrete helps with the configuration definition, generating the XCF file with XML format (section 2.1.5.2).

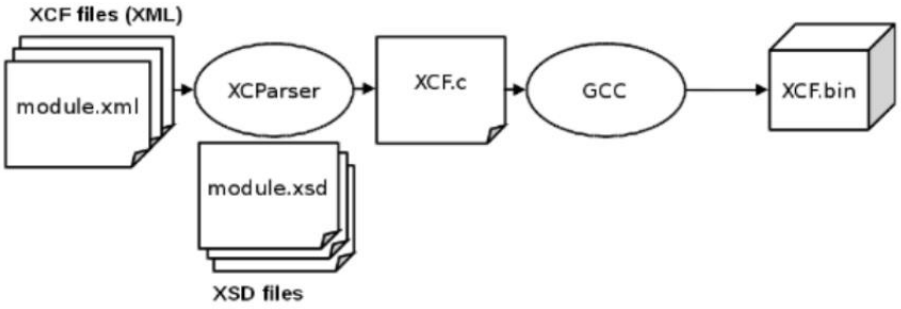

Figure 2-2: XCF file processing steps

By defining the configuration of the hypervisor-based system during the design phase, it is possible to guarantee a fixed allocation of resources for all the components (hypervisor and partitions).

The configuration used can be switched during run-time only by a hypervisor reset request (XResetHypervisor()). Even in this case, the new configuration is provided in the form of a complete allocation, meaning that again a fixed amount of resources is allocated to every component. The process described above can be used to produce a new configuration that can be passed to the XResetHypervisor() service.

The contents of the XCF must not be modified while in use by XNG. This applies to the XCF provided as input when booting the hypervisor and to the ones provided in subsequent hypervisor reset operations (XResetHypervisor()).

Due to the development approach with XNG, the xcparsertool has been developed without targeting any dependability level. Therefore, the output generated by these tools should be

manually verified by the system integrator when targeting higher dependability levels for the system.

It is assumed that the compiler has been credited for the required dependability level and therefore it is expected that the integrator performs this verification at XCF.c level, that is the output generated by the xcparser tool.

2.1.5.1 Xoncrete

2.1.5.1.1 Hypervisor configuration

This subsection describes the hypervisor configuration options. In the XCF generation process Xoncrete use the configuration preferences as input parameters. The more important configuration parameters are related to the strong space partitioning (memory region isolation for each partition), the inter-partition communication and the failure detection, isolation and recovery.

The hypervisor definition is composed of the memory address space, the data area and the entry point description. In the same way, the system hardware needs to be specified to enable its capabilities. The system hardware descriptions includes the memory layout region, the processor, the L2 cache, the board oscillator device and the UART and Console device.

Once the hypervisor and the system hardware are defined the next step is to configure the different partitions and the communication between them. In the same way, the health monitor is configured defining the events and actions.

The final part is the plan configuration and generation, this part is related to the temporal system model, explained in the next subsection.

2.1.5.1.2 Temporal system model

In the classical scheduling theory, a “task” is the execution element that contains all the scheduling attributes: period, deadline, relations with other tasks, etc. Nevertheless, this model does not capture properly the operation of complex systems. Therefore, it will not be used by the Xoncrete tool.

The temporal model and its elements is explained at section 2.3.1 in D3.1.

2.2 The AUTOSAR adaptive platform

This section describes the Adaptive AUTOSAR runtime Platform. Since the platform is quite extensive, we will only give an overview of the existing modules (as of version 20-11). Adaptive Applications (AAs) should use the AUTOSAR Adaptive Platform (AP). The AP is a standardized middleware on top of a PSE51 interface (a single process profile of the POSIX standard). It provides AUTOSAR standardized functionalities to abstract from OS specific APIs. Figure 2-3 gives an overview of the most important AP modules. We will briefly explain them in the following paragraphs. For more details about each module/service have a look at the Adaptive AUTOSAR website: https://www.autosar.org/standards/adaptive-platform/. The picture on the website is similar to the one in Figure 2-3, the boxes directly link to the corresponding module document.

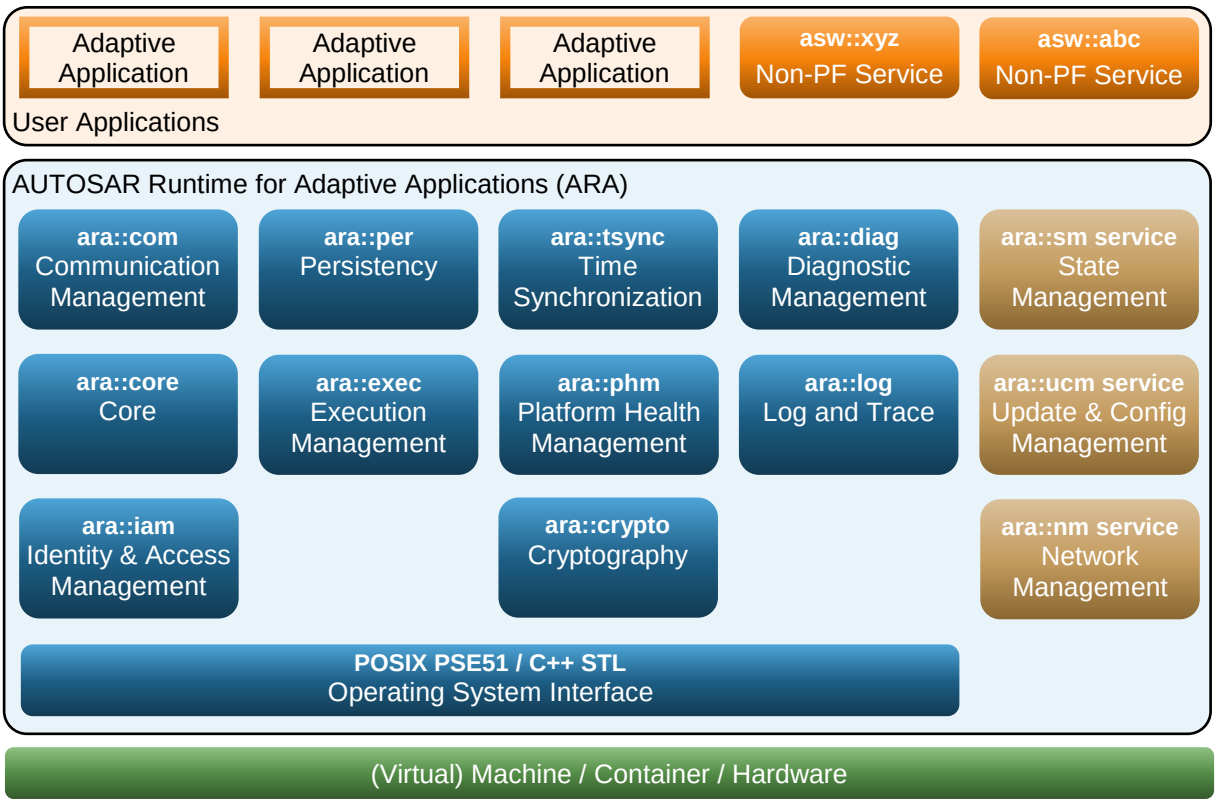

Figure 2-3: Adaptive platform architecture overview

Lifecycles of AAs are managed by the Execution Management (EM). Loading/launching of an application is managed by using the functionalities of the EM, and it needs appropriate configuration at system integration time or at runtime to launch an application. The EM reads the configuration file(s) (called Manifests) of each deployed executable of an application and configures the executables (which are POSIX threads) accordingly. EM will also decide in which order executables are started. The Manifests of AAs contain, for example, the desired scheduling policy (e.g. FIFO, ROUND_ROBIN, …), the priority, or the core affinity. All executables have to report their state to the EM. There are three states the EM expects here: Initializing → Running → Terminating, where the arrows represent the state transitions. The executable may have more states internally, but these three states must be reported to the EM.

Regarding the interaction between AAs, PSE51 does not include IPC (Inter-Process-Communication), so there is no direct interface to interact between AAs. Communication Management (CM) is the only explicit interface. CM also provides Service Oriented Communication for both intra-machine and inter-machine, which are transparent to applications. CM handles routing of Service requests/replies regardless of the topological deployment of Service and client applications.

The State Management (SM) service is somewhat special as it is intended to be ECU-specific, and its implementation is up to the ECU integrator. SM is responsible for all operational states of the AP and may trigger internal state changes based on incoming external events. It may hold multiple internal state machines whose parallel composition constitutes the overall state of the AP. Interaction with AAs shall be realized via CM.

Diagnostic Management (DM) offers diagnostic functionalities standardized in various ISO standards (ISO 14229-5 (UDSonIP), ISO 14229-1 (UDS), and ISO 13400-2 (DoIP)). The configuration of DM is based on classic AUTOSAR Diagnostic Extract Template (DEXT). DoIP is a vehicle discovery protocol and designed for off-board communication with the diagnostic infrastructure (i.e. diagnostic clients, production-/workshop tester). For in-vehicle or remote diagnostics, other transport protocols are often used. UDS is typically used within the production of a vehicle and within the workshops to be able to repair the vehicle.

Persistency offers mechanisms to store information in the non-volatile memory of an Adaptive Machine. The data is available over boot and ignition cycles. Persistency provides standard interfaces to access the non-volatile memory. The Persistency APIs take storage location identifiers as parameters from the application to address different storage locations. Currently, two kind of storage are supported: key-value storage, file storage. Each application may use any constellation of these storage types.

For some applications, Time Synchronization (TS) between different AAs and/or ECUs is of paramount importance. For example, if the correlation of different events across a distributed system is needed, either to be able to track such events in time or to trigger them at a certain point in time. For this reason, a Time Synchronization API exists, so it can retrieve the time information in synchronization with other entities or ECUs.

The Network Management (NM) is based on a decentralized network management strategy. This means, every network node performs activities independently depending only on the NM messages received and/or transmitted within the communication system. The NM algorithm is based on periodic NM messages, which are received by all nodes in the cluster via multicast messages. If any node is ready to go to sleep mode, it stops sending NM messages, but as long as NM messages from other nodes are received, it postpones the transition to sleep mode. Finally, if a dedicated timer elapses because no NM messages are received any more, every node performs the transition to the sleep mode.

One of the declared goals of the AP is the ability to flexibly update the software and its configuration through over-the-air updates (OTA). UCM (Update and Config Management) is responsible for updating, installing, removing, and keeping a record of the software on an AP. Its role is similar to known package management systems like dpkg or YUM in Linux, with additional functionality to ensure a safe and secure way to update or modify the software on the AP.

The concept of Identity and Access Management (IAM) is driven by the increasing need for security, as the AP needs a robust and well-defined trust relationship with its applications. IAM introduces privilege separation for AAs and protection against privilege escalation in case of attacks. In addition, IAM enables integrators to verify access on resources requested by AAs in advance during deployment.

There is an API for common cryptographic operations and secure key management (Cryptography in Figure 2-3). It supports the dynamic generation of keys and crypto jobs at runtime, as well as operating on data streams. To reduce storage requirements, keys may be stored internally in the crypto backend or externally and imported on demand.

The Log and Trace module is responsible for managing and instrumenting the logging features of the AP. Its features can be used by the platform during development as well as in and after

production. These two use cases differ, and the Log and Trace component allows flexible instrumentation and configuration of logging in order to cover the full spectrum. Logging information can be forwarded to multiple sinks, depending on the configuration, such as the communication bus, a file on the system, or a serial console. The provided logging information is marked with severity levels and can be filtered by severity. For each severity level, a separate method is provided to be used by the AAs or other modules. The AP and the logging module are responsible for maintaining the platform stability to not overload the system resources.

The Platform Health Management (PHM) supervises the execution of software. It offers the following supervision functionalities: Alive (checks that a supervised entity is not running too frequently and not too rarely), Deadline (checks that steps in a supervised entity are executed in a time that is within the configured minimum and maximum limits), Logical (checks that the control flow during execution matches the designed control flow), and Health Channel (provides the possibility to hook external supervision results, e.g., RAM test, voltage monitoring, …).

There are a number of core types (in the Core module) defining common classes and functionality used by multiple modules as part of their public interfaces. One of the rationales to define core types was to include common complex data types that are often used in the interface definition.

The AP Operating System is required to provide multi-process POSIX OS capability. Each AA is implemented as an independent process, with its own logical memory space and namespace. Note that a single AA may contain multiple processes, and this may be deployed onto a single AP instance or distributed over multiple AP instances. From the module organization point of view, each process is instantiated by the OS from an executable. Multiple processes may be instantiated from a single executable. Also, AAs may constitute multiple executables. The AP services and the non-platform services are also implemented as processes.

2.3 The AUTOSAR classic platform

The AUTOSAR classic platform (CP) may also play a role in XANDAR. The AUTOSAR XML format for the AP is still changing as the AP is actively developed. The XML format for the CP, however, is rather stable and already supported by many tools for data exchange. Therefore, we may use the CP for data exchange. Currently, it is not planned to setup a CP ECU within the scope of this project, hence, we do not describe the AUTOSAR classic platform in this document.

3 HW and SW monitors

3.1 Model-based generation of fault and operating state monitors

To complement low-level monitors that are designed to observe the platform itself, the toolchain developed in the XANDAR project will support the automatic generation of software targeted at the monitoring of application-level faults and operating states. This software can be seen as an output of the code transformation steps developed in Task 3.2 and will automatically be derived by the software generation framework developed as part of Task 2.5. Therefore, it is highly related to the overall XANDAR design methodology, which was initially introduced in D3.1 and is repeated in Figure 3-1 for convenience.

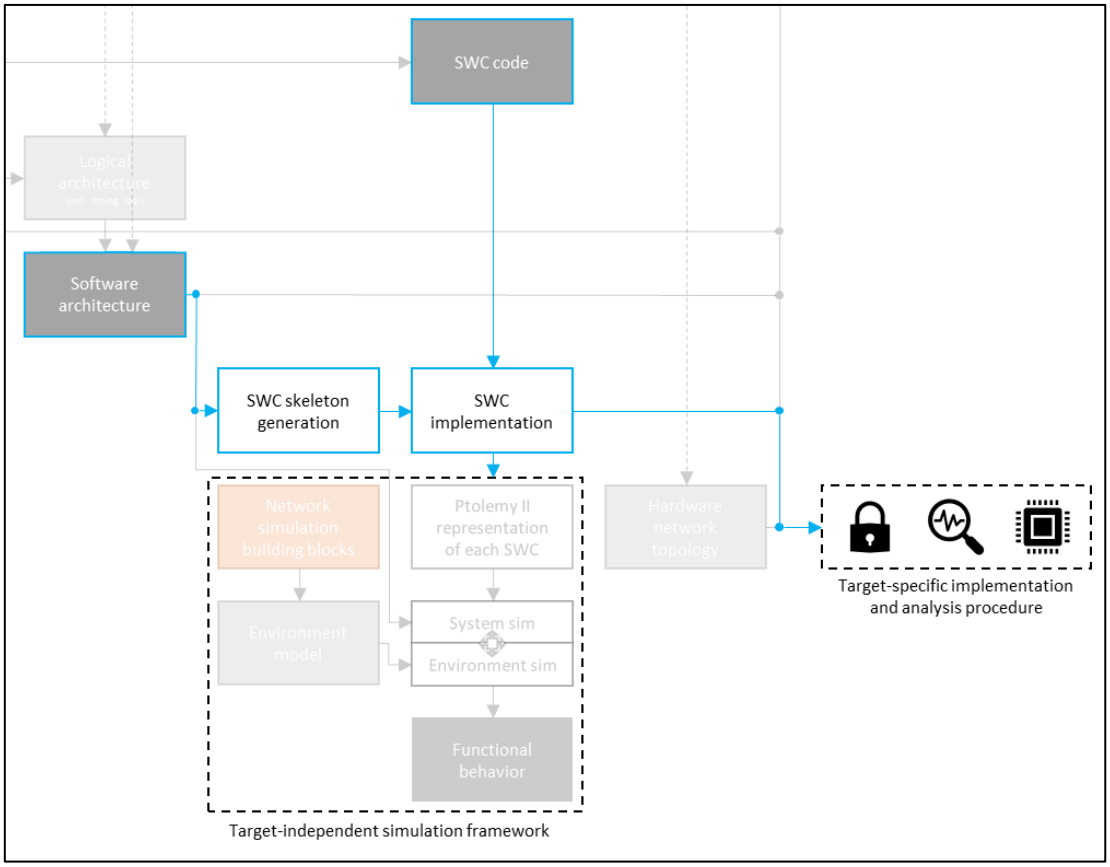

Figure 3-1: XANDAR design methodology as described in D3.1

During the creation of the Software Architecture and the subsequent derivation of SWC implementations, a designer will have the opportunity to specify trigger conditions related to the timing and the value of messages traversing input and output ports of a SWC. An example of a trigger condition is the situation in which an ML/AI item violates its Operational Design Domain (ODD) as described for the avionics use case in D1.1. Another example is the situation in which an internal fault is detected by observing implausible values at the output ports of a SWC during runtime. Designers will have the opportunity to connect suitable reaction sequences to all of these triggers during the derivation of the SWC implementation. During the target-specific implementation steps, suitable monitor software will be integrated into the generated runtime system. A particular target that the relevant code transformation steps developed in Task 3.2 will support are software monitors based on RTLola [5].

3.2 Integration of monitors in the hypervisor

The XNG Health Monitor provides the mechanisms to integrate the automatically generated monitors. This mechanism is composed of the partition events, actions and the logging. Generic health monitor partition events can be generated using the XRaiseHmEvent() service. Architecture-specific partition events can be generated with the above service or map to become generic partition events. Additionally, the XReportHmEvent() service allows a partition to report an HM event, which will be reported in the HM log, in this case the action, assigned to the event, doesn't be executed.

3.2.1 Health monitor configuration

The configuration consists of pairs definition, pairing each event with an action. The pairs are grouped in the HM tables, one table per partition. For easing the configuration and maintainability of the partition's HM table, the XCF defines the multi-partition HM table. The multi-partition HM table can be used for any partition as the default HM table, in this way, the partition only has to define the elements not defined in the previous table.

The XML XCF snippet below shows an example of use of the multi-partition HM tables. The multi-partition HM table is used by the partition as its default HM table. In the case of the hardware fault event, the default action is redefined as a coldResetPartition action.

```
<MultiPartitionHmTables .../>
	<MultiPartitionHMTable name="multipartitionHmTab0">
		<Event name="scheduleError" action="haltPartition" />
		<Event name="numericError" action="haltPartition"/>
		<Event name="stackOverflow" action="haltPartition"/>
		<Event name="applicationError" action="haltPartition"/>
		<Event name="internalError" action="haltPartition"/>
		<Event name="illegalRequest" action="haltPartition"/>
		<Event name="memoryViolation" action="haltPartition"/>
		<Event name="hardwareFault" action="haltPartition"/>
		<Event name="powerFail" action="haltPartition"/>
		…
	</MultiPartitionHMTable>
</MultiPartitionHmTables>
<Partition name="Partition0" ..>
	…
	<HmTable baseHmTable="multipartitionHmTab0" nbHmLogs="64">
		<Event name="hardwareFault" action="coldResetPartition"/>
	</HmTable>
	…
</Partition>
```

4 Compiler selection for source-to-source transformations

This section describes our initial investigation of the various source-to-source transformation tools that can be employed in the XANDAR toolchain. While there are many tools available and as part of this analysis, we concentrate on the most representative tools that can be useful in the XANDAR toolchain (in particular those transformations can be applied in the "Safety/Security pattern application" and "Code generation" of the XANDAR development process Figure 2-1, in D2.1). In general, the source code transformations provide a variety of capabilities in a toolchain and the target is to automatically generate software code (at source level) that is characterized by specific properties.

4.1 Pluto

Pluto [14] is a source-to-source transformation tool that is based on the polyhedral model. The polyhedral model is used to provide an abstraction, so high level compiler optimizations, such as loop-nesting, can be performed. Also, Pluto provides automatic parallezation for C programs and automatic generation of OpenMP code. Other optimizations provided by the Pluto tool are automatic generation of specific tile sizes, loop unrolling factors, and outer loop fusion structure. In order to use Pluto on a sample code, the tool must be able to extract the polyhedral model from the source code. Below we provide the command line usage of Pluto after installation:

- `./polycc <source_file.c>`, basic usage, extract polyhedral model
- `./polycc -parallel <source_file.c>`, automatic parallelization

In order to apply specific optimizations, the following command-line command can be used:

- `./polycc -tilled <source_file.c>`
- `./polycc -tilled -parallel <source_file.c>`

The code to be transformed in the source file must be between '#pragma scop' and '#pragma endscop', as depicted in the following snippets.

Source code before transformation

```
#pragma scop
     for (i=0;i<n;i++)
          for (j=0;j<n;j++)
               B[i][j] = A [i][j] + u1[i]*v1[j] + u2[i]*v2[j];
     for (i=0;i<n;i++)
          for (j=0;j<n;j++)
               x[i] = x[i] + beta* B[j][i]* y[j];
      for (i=0;i<n;i++)
          x[i] = x[i] + z[i];
     for (i=0; i<n;i++)
          for (j=0;j<n;j++)
               w[i] = w[i] + alpha* B[i][j]*x[j];
#pragma endscop
```

Source code after transformation

```
if (N>=1) {
     for (t2=0;t2<N-1;t2++){
          lbv=0;
          ubv=N-1;
#pragma ivdep
#pragma vector always
          for (t3=lbv;t3<=ubv;t3++){
               B[t2[t3] = A[t2][t3] + u1[t2]*v1[t3] + u2[t2]*v2[t3];
               x[t3] = x[t3] + beta*B[t2][t3]*y[t2];
           }
      }
      lbv=0;
      ubv=N-1;
#pragma ivdep
#pragma vector always
     for (t2=lbv;t2<ubv;t2++){
          x[t2] = x[t2] + z[t2];
      }
      for (t2=0;t2<N-1;t2++){
          for (t3=0;t3<N-1;t3++){
               w[t2] = w[t2] + alpha* B[t2][t3]*x[t3];
           }
      }
}
```

4.2 GeCoS

GeCoS [15] is a C compiler infrastructure written in Java which follows the Model Driven Engineering design principles. Its name stands for “Generic Compiler Suite” but its main usage is to act as a source-to-source Compiler. GeCoS applies various features on a given source code, such as an intermediate representation of the code, SSA representation and DAG representation. There are also some GeCoS scripts which implement the compiler transformation. Some of them include loop unrolling, procedure inlining and array scalarization. To invoke the GeCoS scripts, a GS prefix must be placed in the source code with an associative ‘pragma’.

Below is an example of automated loop unrolling using a GS prefix:

Source code before loop unrolling

```
#define N 2
int foobar() {
     int i,j;
     int t[N][N];
     #pragma GCS_UNROLL
     for (i=0;i<N;i++)
           #pragma GCS_UNROLL
           for (j=i;j<N;j++){
                t[i][j] = i == j ? 1 : 0;
           }
}
```

Source code after loop unrolling

```
int foobar(){
     int i;
     int j;
     int [2][2];
     t[0][0] = 1;
     t[0][1] = 0;
     j =2;
     t[1][1] = 1;
     j = 2;
     i = 2;
}
```

Intermediate representation graph: GeCoS creates a graph of the program’s basic blocks and their connection, providing a detailed representation of the source code’s execution flow.

Intermediate representation structures: The GeCoS IR creates abstraction layers in the source code. The fundamental ones are Blocks for high level constructs (if, for, while, etc), instructions to model low level instructions, symbol to model objects and procedures for functions and other.

4.3 LLVM

The LLVM project is a widely-used collection of modular and reusable compiler and toolchain technologies. Concerning source-to-source transformations LLVM provides one basic ways to apply these kind of transformations. First, through developing front-end standalone tools using the Libtooling library. Libtooling is an easy API library for the frontend part of the LLVM. Also, by combining this with the LLVM/Clang source code as included libraries, someone can develop his own parsing and source transformation tools. The main LLVM classes for this purpose are the `RecursiveASTVisitor` class and the Rewriter class. What we can basically do with these, is to extract information from the ASTree of a source code using the `RecursiveASTVisitor` class and then write back to our source code using the Rewriter API.

Note that the Clang's AST is immutable, so we can only extract information and not apply the actual changes to it.

LLVM provides many other useful capabilities at different levels during the compilation. A good example is the compiler optimizations and transformations at intermediate representation level through LLVM passes.

Generally speaking, LLVM is widely used tool with a big open community that acts as contributors. Pluto is actually based on LLVM/Clang (front end of LLVM tool) for applying the various transformations. A combination of Pluto and LLVM is our proposal for the XANDAR project, but more work is needed in order to take into consideration various other parts of the project.

5 Code transformations for reliability enhancement

As the process technology continues to shrink, a large number of SRAM bit cells in on-chip caches is expected to be faulty [1][2]. This problem is particularly pronounced in on-chip caches for two main reasons. First, an increasing larger portion of the chip area is devoted to caches. Second, the cache memory arrays are built with minimum sized, thus more prone to failure, SRAM cells. Most importantly, the current trends towards near threshold execution simply exacerbate the cache reliability problem [6]. As a result, many Cache Fault-Tolerance (CFT) techniques have been developed to ensure error free execution in the presence of memory faults. This is a widely studied area where many alternative CFT organizations have been proposed. However, the impact of compiler transformations in the performance of faulty caches has been largely disregarded from the research community and the industry either when the target is the average or the worst-case performance. This research direction will be mainly explored as part of WP3 (and in particular in Task 3.2) of XANDAR project.

Modern compilers support a large number of code transformations which directly impact code footprint, performance, and power efficiency. Our current activities concentrate on showcasing that compiler transformations can be an effective mean to boost the fault-tolerance capacity of the memory system against hard-errors and most importantly without compromising the L1 cache latency. The proposed techniques will be initially investigated in the context of faulty level-1 instruction and data caches in which the block disabling technique is employed.

More specifically, our plans are as follows: Firstly, we will identify the main compiler transformations that affect performance when faulty caches exist. Secondly, we will show how these optimizations and their parameters affect the cache and system performance. Thirdly, using detailed simulations, various linear algebra kernels (and the computational kernels of XANDAR use-cases), a wide range of fault-probabilities, and a plethora of cache fault maps, we expect to show that the proposed approach is able to offer significant benefits in cache performance, especially when the positions of the defective cache blocks are exposed to the compiler.

In order to evaluate the effect of various code optimizations, Figure 5-1 shows the gathered statistics for bicg benchmark (example computational kernel) from Polybench/C suite assuming 16KB first level data cache (DL1) size and 8KB first level instruction (IL1) cache size. In the primary y axis blue (green) bars show the DL1 (IL1) misses. Furthermore, in the secondary y axis, the orange line depicts the number of execution cycles for each case. The numbers in the y-axis are the normalized values with regards to the O0 code optimization.

More specifically, Figure 5-1 shows a sub-set of the code optimization that we intent to study. For example, in DL1 case we evaluate the following code optimizations: 1) register blocking 2/4/7, 2) vectorization, 3) scalar replacement and 4) O1/2/4/s. In the IL1 case, we start with the loop unrolling transformation assuming various unroll factors. In order to isolate each specific code optimization, we have manually applied each optimization over the vanilla source code of each benchmark version. The gcc optimizations (O1/2/3/s) produced using the appropriate compilation flag. Figure 5-1 clearly shows that different code optimization may have significant impact in the number of misses and also in the execution time of the application. Code optimizations may affect the code size and/or the register allocation, which may lead to performance fluctuations. Moreover, in case of computationally intensive applications, the

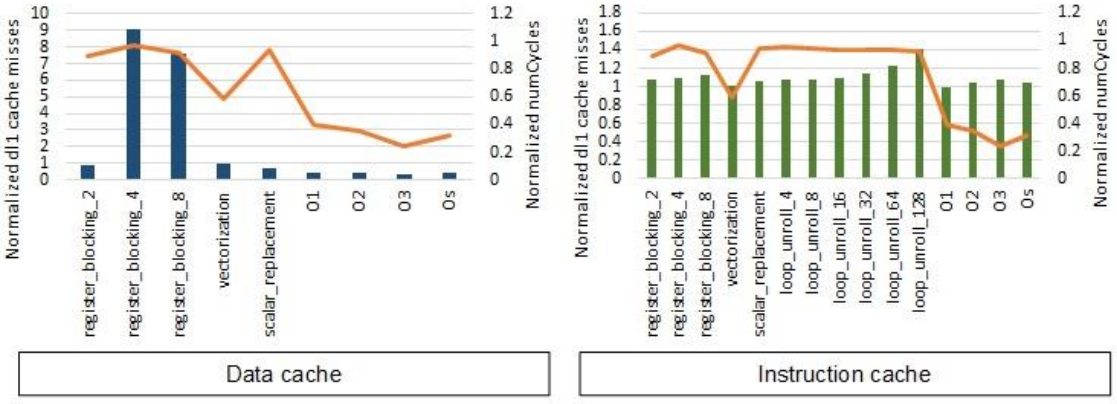

Figure 5-1: Normalized cache misses and numCycles (wrt. -O0 optimization parameter)

studied code optimizations can effectively reduce the DL1 cache pressure. A first conclusion is that based on the specific cache configuration (cache size, associativity) there is a unique combination of optimizations that may boost the cache performance.

In addition, Figure 5-1 shows some clear trends in fault free caches. Starting from the register blocking cases, it seems that as the register blocking is extended to higher register blocking factors, the number of misses in both cases (DL1 and IL1) are increased with respect to -O0 case. Register blocking increases the code side, thus the number of L1 misses. Similar effect has the loop unrolling in IL1s as the unrolling factor increases. Finally, as figure indicates from performance perspective, the combinations of code optimizations that comes with the O1/2/3/s choices are the best choice.

Moving to caches which contains malfunctioning cells, Figure 5-2 presents the normalized DL1 cache misses (blue bars, primary y-axes) and the numCycles (orange line, secondary y-axes) of the various depicted code optimizations with respect to O0. All the statistics are averaged values across the total number of simulations assuming different cache fault maps (in each fault map the locations of the hard-errors is randomly generated). We have to mention here that we assume that caches are equipped with the Block Disabling scheme [10] in order to deal with the presence of malfunctioning memory cells. Moreover, each bar, equipped with the positive and negative error bars which indicate the best and the worst reported DL1 misses for a specific fault map. Finally, Figure 5-2 is divided vertically into three graphs. Each graph corresponds to different probability of failure (pfail) for each memory cell, which lead to different number of malfunctioning cells.

Figure 5-2 shows various interesting results. First, as the error bars indicate the position of the faulty cells (fault maps) have significant impact on cache misses, this phenomenon is

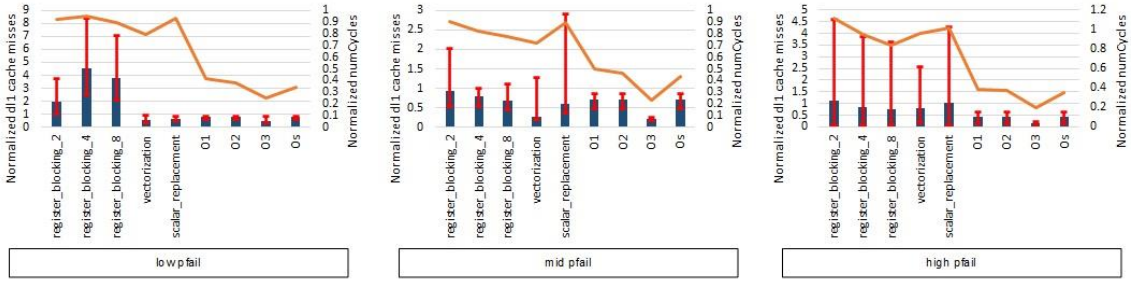

Figure 5-2: DL1 cache statistics for three different pfail situations

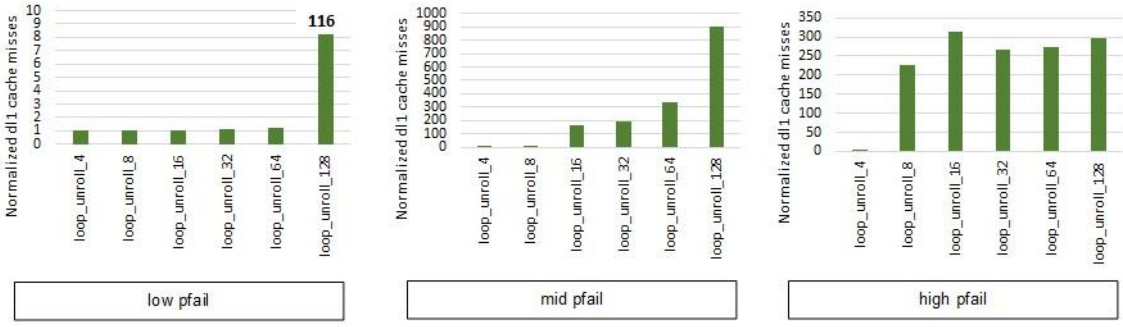

Figure 5-3: IL1 misses for various loop-unrolling factors and three pfail situations

expanded as we move into higher pfails. This outcome shows that cache performance is not uniform in the presence of faults. Another interesting observation can be extracted from the Figure 5-2 if we focus on the register blocking optimization across the three different pfails. As figure depicts, the trend in the case of register blocking is inversed (in terms of cache performance) as the number of malfunctioning cells increased. In low pfail (left most graph) the register blocking with factor 2 exhibits the lower number of DL1 misses, while in mid and high pfail we can see that the higher register blocking factor can lead to better cache performance. The reason for this behaviour is that the register blocking reduces the number of memory references (load/ store instructions), so it manages to reduce the pressure of the faulty DL1.

Similarly to the previous cases, Figure 5-3 shows the reported IL1 misses when loop unrolling is applied with different unrolling factors. It is clear that the loop unrolling factor is bounded from the instruction cache size. A significant impact of Block Disabling scheme that is applied in this case to deal with the malfunctioning memory cells is that a significant memory space is disabled. It is obvious that IL1 size is reduced as the pfail increases. As a result, the loop unrolling factor should be adjusted for each specific pfail scenario.

To sum up, the above analysis presents a first-class justification that compiler optimizations can have a significant impact on the system performance, when system operates in the presence of faults. Therefore, software/release engineers should consider the pfail conditions under which the system will be operate and adjust the code optimizations accordingly. As noted, this initial study will be further extended in Task 3.2 taking the XANDAR applications as input.

6 Security extensions

6.1 Need for security artefacts

The mixing of critical and non-critical software component brings significant advantages within the networked embedded system such as consolidation of different functionalities within a single system, reducing costs, and increasing efficiency. But at the same time, these mixed-criticality software brings major security challenges due to sharing of critical system control and data [7][8][11]. The advanced embedded multiprocessor System-on-Chip architectures allow implementation of mixed-criticality software applications and provide on-chip security mechanisms to segregate and protect system resources, right from the booting of the system, to prevent misuse and compromise. However, these security and safety measures have been found vulnerable due to a lack of secure software design practices and the adoption of ad-hoc or passive defences [3][4][9].

To approach this, an X-by-Construction approach has adopted in XANDAR. The idea of the said approach is to bake-in/incorporate security defences across various layers of the system hardware/software architecture of the overall software system of the networked embedded system.

6.2 Purpose of security artefacts

AUTOSAR is an initiative that standardise software architecture of automotive Electronic Control Units (ECUs) by increasing reuse and ex-changeability of software modules between vehicle manufacturers and suppliers. One of the objectives of AUTOSAR is to decouple the hardware-independent application software from the hardware-dependent software (memory drivers, crypto drivers, communication drivers etc.) by employing a run time software abstraction.

AUTOSAR ADAPTIVE software system architecture is based on object-oriented approach which offers a crucial advantage. As applications and services can be integrated into the system at runtime, which means that they can be developed, tested and distributed or updated independently of one another. This enables realisation of adaptive automotive ECUs, making it possible to update applications over a vehicle’s entire life cycle and add/manage new software functions later as necessary.

To enhance the security, a range of security artefacts and system-level controls will be incorporated into the XANDAR platform. The objective of these safety artefacts is to establish a means to regulate/monitor software function-level activities at its entry/points closer to the source as illustrated in Figure 6-1.

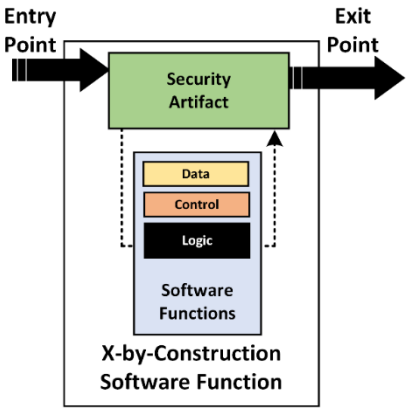

Figure 6-1: Conceptual diagram of X-by-construction software function

All the control and data traffic transmitted to and received by the software function from the other platform/system software layers shall pass through this controlled and monitored interface. The realisation of security artefact can be either in hardware or software will be subject to further research in XANDAR.

6.3 Security requirements & artefacts

Based on XANDAR application use case (D1.1), the following are the identified non-functional security requirements:

- Data confidentiality (at rest, in motion)
- Data integrity (at rest, in motion)
- Authentication & authorization (firmware, OS, partition, software applications)
- Trusted execution (software compartmentalisation, access control)
- Runtime security monitoring

There is a need for various platform and system level components and defences such as Root-of-Trust, Secure Boot, Trusted Platform Module (TPM) or Crypto processor, Secure Debug, Secure Update etc.) to implement the identified security requirements. These system-level components and defences can be realised in software, hardware or both, the XANDAR software system shall provide layered support to securely program, manage and control the underlying supported features during the lifecycle of the platform.

6.4 Secure system/platform lifecycle management

Based on the defence-in-depth security model, it is envisaged that XANDAR hardware platform shall go through a series of execution stages to achieve secure platform lifecycle management. The following are the envisaged platform execution stages:

1. Secure Boot
2. Check for Platform Security Policies
3. Secure Configuration & Provisioning
4. Secure Application Software On-boarding
5. Runtime Security Monitoring

Further research is required to identify and refine the in-depth details of each execution stage, to evaluate their impact on the functional requirements and to justify their suitability within the scope of XANDAR project.

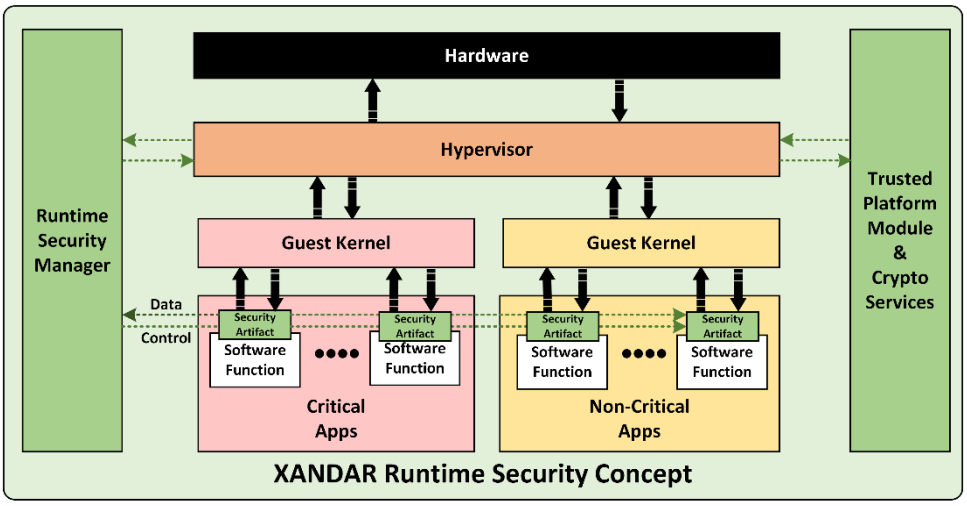

Figure 6-2: The XANDAR run-time security concept

6.5 Blueprint of platform/system-level runtime security

The objective of XANDAR runtime security concept is:

1. To build granular resource isolation and establish trust boundaries among mix-critical system components. The design choices made will be based on the application use-case driven system security model in-line with the overall security objectives.
2. To gather localised valuable runtime insights of the system and software components and leverage this localised insight to build a holistic overall context of the runtime system.

Figure 6-2 shows the conceptual diagram of XANDAR runtime security concept. This runtime security concept requires:

1. **Security Artefact –** which will be deployed at the interfaces (entry/exit checkpoints) of system components, serving as runtime gatekeepers to enforce security and maintain appropriate defences.
2. **Runtime Security Manager –** will serve as centralised master control of distributed *Local Security Monitors.* It shall be responsible for the overall management and status monitoring to build a holistic runtime system context.

The design decisions (the level of abstraction and granularity), their interfaces and in-depth implementation (either in hardware, software or both) details of the proposed XANDAR runtime security concept is not finalised and subject to further research during the XANDAR project.

7 Support for ML/AI applications

As noted in section 3.2.4 of D3.1, the inference (run-time) part of the neuromorphic applications is handled as a regular SWC. The inference part of the neuromorphic applications will be extracted from a high-level AI/ML framework as a standalone SWC in C/C++ programming language and will be fed to the XANDAR toolchain as a typical software implementation with specified input and output ports. In this way, even the neuromorphic SWC can be wrapped into a Ptolemy II actor that can be used to simulate its behaviour in the specific environment.

However, from the perspective of the design methodology, there is one main difference of neuromorphic applications wrt. other SWCs. In particular, each neuromorphic SWC will be provided with different functional properties. While the main topology of the given neuromorphic applications will be kept the same, we will provide different implementations of each neuromorphic SWC by varying its data type. This means that each neuromorphic SWC will be provided in at least two data types e.g., FP32, FP16, INT32, and INT8, thus with different accuracy/prediction capabilities. The suitable implementation will be decided by the functional analysis and verification phases of the XANDAR toolchain.

While the previous description pertains to the functional modelling of the neuromorphic applications, the run-time management of the AI applications will be managed by the system wide End-to-End-Flow model followed by the XNG hypervisor. In particular, the XANDAR AI applications will follow all the skeleton and communications artefacts described in section 2.1.5.2.

8 Interfaces with the V&V framework

Since a number of software/hardware components and platforms will be designed and implemented in WP4, proper V&V of those components and platforms and also embedded software to be developed using those components and platforms. Thus, there will be a close interaction between WP5 and WP4.

Meta-V&V activities (Task 5.4) of the toolchain components and platforms to be developed in WP4 will be conducted (e.g., security platform for trusted embedded devices). Also, V&V of the embedded software to be developed using those components and platforms will be conducted.

As discussed in D5.1, meta-V&V activities will be implemented and conducted by developing automated test suites in unit, integration, and system testing levels for the XANDAR toolchain, including the components and platforms. Also, static V&V activities (including inspections) will be conducted to ensure delivering defect-free components and platforms in WP4. Given the nature of certain deliverables (e.g., Software System Specification for Trustworthy & Secure Computing Platforms in Task 4.1), only static V&V will be possible (a main approach will be inspections [12]).

Some other tasks (such as Task 4.3: Platform Health Monitoring and Task 4.4: Run-time Security Monitoring) are of V&V nature, i.e., monitoring is by definition a V&V approach [13]. Thus, there will be a close interaction between WP5 and WP4.

9 Summary

In summary, this deliverable presents the work performed as part of Task 4.1 (Software System Specification for Trustworthy & Secure Computing Platforms) of WP4 of XANDAR project. The target was to specify the XANDAR run-time system (consisting of the XNG hypervisor and the AUTOSAR adaptive) as well as the envisaged online monitoring, self-adaptation and self-healing mechanisms. In addition, as part of this task, we present a tentative plan for the proposed code transformations for reliability enhancements and our analysis for a source-to-source code transformation framework. Furthermore, specific security extensions of the XANDAR platform are also described. Finally, consider the support for ML/AI applications, our plan is to handle them as regular software components that be orchestrated by the system wide End-to-End-Flow model followed by the XNG hypervisor.

References

[1] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V.De. Parameter Variations and Impact on Circuits and Microarchitecture. Proc. of Design Automation Conference, 2003.

[2] K. Bowman, J. Tschanz, C. Wilkerson, S.L. Lu, T. Karnik, V. De, and S. Borkar. Circuit Techniques for Dynamic Variation Tolerance. Proc. of Design Automation Conference, 2009.

[3] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19). Association for Computing Machinery, New York, NY, USA, 1741–1758. DOI: https://doi.org/10.1145/3319535.3363206B. Finkbeiner, S. Oswald, N. Passing und M. Schwenger, „Verified Rust Monitors for Lola Specifications," in *20th International Conference on Runtime Verification (RV 2020)*, 2020.

[4] D. Cerdeira, N. Santos, P. Fonseca and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems," 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1416-1432, doi: 10.1109/SP40000.2020.00061.

[5] B. Finkbeiner, S. Oswald, N. Passing und M. Schwenger, „Verified Rust Monitors for Lola Specifications," in 20th International Conference on Runtime Verification (RV 2020), 2020.

[6] S. Ganapathy, J. Kalamatianos, K. Kasprak, and S. Raasch. On Characterizing Near-Threshold SRAM Failures in FinFET Technology. Proc. of Design Automation Conference, 2017.

[7] M. Hagan, F. Siddiqui and S. Sezer, "Enhancing Security and Privacy of Next-Generation Edge Computing Technologies," 2019 17th International Conference on Privacy, Security and Trust (PST), 2019, pp. 1-5, doi: 10.1109/PST47121.2019.8949052.

[8] A. Kott and P. Theron, "Doers, Not Watchers: Intelligent Autonomous Agents Are a Path to Cyber Resilience," in IEEE Security & Privacy, vol. 18, no. 3, pp. 62-66, May-June 2020, doi: 10.1109/MSEC.2020.2983714.

[9] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. ACM Comput. Surv. 51, 6, Article 130 (February 2019), 36 pages. DOI:https://doi.org/10.1145/3291047

[10] G. S. Sohi. Cache Memory Organization to Enhance the Yield of High-Performance VLSI Processors. Trans. on Computers, 1989.

[11] F. Siddiqui, M. Hagan and S. Sezer, "Establishing Cyber Resilience in Embedded Systems for Securing Next-Generation Critical Infrastructure," 2019 32nd IEEE International System-on-Chip Conference (SOCC), 2019, pp. 218-223, doi: 10.1109/SOCC46988.2019.1570548325.

[12] https://www.sciencedirect.com/science/article/pii/S0065245808604842

[13] https://ieeexplore.ieee.org/abstract/document/5966590

[14] http://pluto-compiler.sourceforge.net/

[15] https://gitlab.inria.fr/gecos