Journal of Universal Computer Science, vol. 27, no. 8 (2021), 894-911
submitted: 8/4/2021, accepted: 2/5/2021, appeared: 28/8/2021 CC BY-ND 4.0

A Compiler and Language Support for Designing Mixed-Criticality Applications

Nermin Kajtazović
(Siemens AG, Graz, Austria
https://orcid.org/0000-0003-1892-5100, nermin.kajtazovic@siemens.com)

Peter Hödl
(Siemens AG, Graz, Austria
peter.hoedl@siemens.com)

Leo Happ Botler
(Siemens AG, Graz, Austria
https://orcid.org/0000-0002-4683-2422, leo.happ_botler@siemens.com)

Abstract: Coexistence of software components and functions of different criticality in a single computing platform has challenged the safety community for the past two decades. Despite efforts that have been made so far, dealing with mixed-criticality has still left some room for improvements. One particular concern here is that partitioning of hardware and software resources with regard to criticality (safety related, non-safety related) has direct implications on how safety measures need to be realised. For example, a self-test that must meet certain diagnostic coverage for the microcontroller core by inspecting its instructions, needs to cover only those instructions which are able to affect a safety function. Available software mechanisms and tools are to a certain extent still unable to deal with such a fine-grained selection of resources. In this work, we introduce a compiler extension and language support which enable accurate selection of data based on their criticality. The compiler extension serves to establish detailed traceability between the software code and its representation in runtime memory. With the language support, the individual data elements can be classified based on the desired safety integrity level. As a result, safety measures that operate on data (e.g. Abraham test for SRAM [Nair et al., 1978]) can achieve better coverage. The method has been evaluated and applied to industrial safety controllers. We provide here relevant performance figures and discuss possible applications of the method in other fields.

Keywords: Mixed-Criticality, Compilers, Code Analysis, Memory Analysis
Categories: C.3, B.1.4, D.1.5, D.2.10, D.3.4
DOI: 10.3897/jucs.71831

1 Introduction

Mixed-criticality systems consolidate functions that must meet different non-functional requirements, such as security, safety and availability [Burns and Davis, 2017, Simó et al., 2021]. For embedded systems that can be found today in automobiles, industrial automation, healthcare and other fields where functional safety is a primary goal, *mixed-criticality* mainly refers to coexistence of system functions that implement different levels of quality assurance, or safety integrity acc. to IEC61508 [Esper et al., 2015].

In order for such systems to achieve a desired level of assurance, it is of utmost importance to ensure that lower criticality functions cannot dangerously affect systems' integrity – for example, that a non-safety related software component inadvertently activates an actuator (e.g. an electric valve [Hardy, 2010], p.64) by overwriting data in safety-related memory regions of the microcontroller.

Industrial safety standards provide here a solid body of rules: it must be guaranteed that higher criticality functions are always timely served, get enough computing resources and that resources they manage cannot be altered by lower criticality functions. Due to a wide variety of system architectures available today, different ways exist on how these requirements can be addressed. Moreover, this diversity has also an impact on how the system architecture can be formed with regard to criticality. In complex computing platforms with integrated hardware virtualisation or multi-core capability for instance, real-time operating system (RTOS) and applications can be executed in isolated environments (partitions). Some sort of a temporal program monitor and scheduler ensure that higher criticality partitions always get required computing resources, whereas accesses to their internal state are controlled by means of memory protection. Thus, the criticality is assigned here on the level of a complete software architecture within a partition. On the other side, there is a class of RTOS-based or bare-metal microcontrollers where no such hardware support is available, and for those systems, ”partitioning” must be carried out between pieces of code and data within a single physical address space.

As reported in several studies [Ebert and Jones, 2009, Ebert and Favaro, 2017, Traub et al., 2018, Broy, 2018, Mössinger, 2010], embedded software is becoming gradually complex and the usage of object-oriented languages with advanced runtime environments is a common practice today. This development has also made it difficult to properly conduct fine-grained separation of software elements (code, data). In particular, for the above mentioned system model, ensuring spatial isolation is of more concern here, because means to reliably identify (and thus to classify) all required software elements are limited. Currently, compiler-enforced memory sectioning is used for this purpose [Autosar, 2017]: the runtime memory layout is organised in a way that pieces of code and data can be allocated to specific memory regions; the criticality is then assigned to these memory regions.

1.1 Problem Statement

Compiler-enforced memory sectioning has a limited quality with regard to elements it can identify. As pointed out by Brauer et al. [Brauer et al., 2015], representation of the software behaviour in runtime memory differs to what we have in the source code. More specifically, as we’ve shown in our previous work [Kajtazović et al., 2020], that compilers may, in the course of the compilation process, create various housekeeping data and managing code which are part of the software runtime environment for our RTOS, applications. These artefacts are interlinked with the rest of software in a way that, in some situations, correct functioning of the overall system may depend on their integrity. The main issue is, however, that they cannot be reliably identified and thus classified based on criticality.

We narrow this problem by considering the following example:

```
//——————————————————————————————————————————
DeviceManager& device(void)
{
  // lazy construction of the instance
  static DeviceManager deviceManagerInstance;

  return deviceManagerInstance;
}
```

The code snippet here shows a lazy initialisation of a sample class *DeviceManager*. For the sake of clarity, we may assume that this class provides the central lifecycle management services for a safety controller, for instance for the battery management system in an electric/hybrid vehicle [Macher et al., 2015].

Now, we may further assume that maintaining the safe state is realised here:

```
...
  // execute a controlled device shut down
  device().enterSafeState();
```

Reflected to the battery management system, definition for the safe state could be that the vehicle must be de-energised from the main power supply. Software implementation above could for example achieve that by controlling some general purpose I/Os on the board of the controller. Now, in order for this method to succeed, correct functioning of the language runtime (in this case C++) is essential. This is because of the fact that the call to the *enterSafeState()* method involves interaction between software and various objects of the C++ runtime.

Inability to identify and thus to classify those parts of the language runtime according to criticality has the following consequences: (**a**) unprotected data fragments pose a threat to functional safety since one single failure may dangerously affect the system's safety function; (**b**) "inadvertently" protected data may reduce system's availability (e.g. a self-test shuts down a system because of a data integrity failure in non-critical memory regions). The former is, obviously, more of concern here, since such unidentified failure modes could result in incomplete coverage by safety measures.

1.2 Method and Contributions

In this work, we introduce a method that allows to identify all software elements in runtime memory and to classify them based on criticality. The identification is achieved by recovering traceability links between the source code and objects in memory during the standard compilation process. The classification based on criticality is applied to objects in memory based on user-provided language annotations in the source code. In short, accurate selection and classification of software elements can help to achieve better coverage of safety measures that operate on data/memory.

This work is based on our original paper [Kajtazović et al., 2020] and extends it as follows:

- Language support to assign different criticality levels to software elements is introduced in Sec. 3.2.
- Details on recovery of C++ runtime objects, where assembler code simulation is used, are described in Sec. 3.1.2.

– Discussion about the application of the method in other domains/fields and for other purposes has been provided in Sec. 5.

1.3 Postulate

The method we introduce here is applicable to projects where the source code is available – at least for software components for which the memory layout shall be analysed. Further, the system model we assume here is a typical bare-metal or RTOS-based embedded system, on which software is installed as a single executable. Finally, we restrict our analysis to the embedded subset of the C++ language (i.e. no runtime type information (RTTI), no multiple inheritance, no exception handling, no dynamic memory management).

2 C/C++ Memory Model: Analysis

The memory model of a language defines how the code and data shall be organised at runtime and how the compiler shall create the necessary runtime system based on language features (e.g. inheritance), if one is required [Hathhorn, 2012, Rosenblatt, 1993]. For the C++ language (and also for C, but in some reduced fashion), the memory model requires to create a runtime system with a number of data artefacts needed for managing the code and data (cf. Stroustrup's C++ object model in [Lippman, 1996]). For example, for all global class instances the compiler needs to build a list of pointers to their constructors in order to create and initialise instances at startup. For initialisation, a special code that executes the construction of instances upon startup needs to be setup and linked as part of the runtime system.

2.1 Structure

An overview of the C++ memory model is given in Fig. 1. To demonstrate the usage of all data artefacts in memory, our static instance of the *DeviceManager* example is used. Further, it is assumed here that this class derives some functionality from the base class *LifecycleManager* to provoke generating a virtual function table. Note that the generation of data artefacts depends on a few properties of the instances/variables, such as duration (static, non static), scope (global, file, method) and on complexity of the software design (i.e. whether routing through a class hierarchy is required or not).

The view in the middle shows what is being generated in memory for the variable *deviceManagerInstance*. This is a static memory view that can be found in binary object and executable files, where memory regions are enriched with symbolic information such as names of source code variables and files, and some other metadata required to load the file into memory (cf. ELF file format specification).

In the physical address space, each of the data artefacts from the symbolic view is mapped to a dedicated memory region. (This is, in fact, a structure which is created after a binary file has been loaded into memory).

2.2 Data Artefacts in Memory

Generally, three parts of the C++ runtime system are responsible for managing code and data: (**1**) global, (**2**) instance-related and (**3**) type related binary objects . Global and

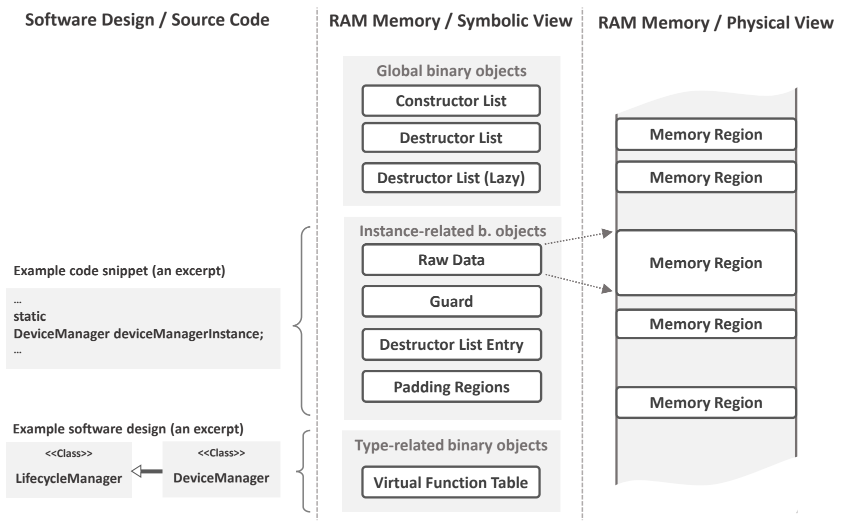


Figure 1: Overview of the C++ memory model: example code that is used here to demonstrate the structure of the generated memory layout ***(left)****; symbolic view of RAM memory with all data artefacts created for deviceManagerInstance* ***(middle)****; physical view of RAM memory where each of the data artefacts is mapped to dedicated memory regions* ***(right)****.*

instance related binary objects are responsible for lifecycle management of instances/-variables (i.e. their initialisation and removal). The latter are, in contrast, created for each instance/variable. Type related binary objects are created for some types, in particular classes with virtual methods, and are responsible for routing calls through the class hierarchy.

The message from Fig. 1 is that even for a single instance/variable, the compiler may create a set of binary objects and thus reserve multiple dinstinct regions in memory:

Raw data (variable/instance). This is a pure content of a variable/instance; in the example above, it is the content of the local static variable *deviceManagerInstance* without any parts of the C++ runtime system.

Constructor and destructor list. Global constructor and destructor list objects are created for managing the initialisation and removal of global class instances. The former is used by the startup code to initialise the global class instances by calling their constructors, the latter is used by the shut down code (which is also generated by the compiler) for calling the corresponding class destructors.

Destructor list for lazy initialisation. Destructor list is a special data structure which is created to support removal of objects which are lazy initialised. Lazy initialisation is a feature in C/C++ that allows instances to be created on demand, instead of being created at startup. There are different purposes of using lazy initialisation, for example to reduce the startup time or to use an instance only if certain condition is met. Our *device-ManagerInstance* is an example of lazy initialisation. During compilation, a registration entry is created in the list for the instance and is used by the shutdown code for ordered destruction.

Guard variable. The guard variable is created for local static instances/variables in method scope to support their lazy initialisation. The behaviour of the guard is similar to what we achieve when using a singleton idiom [Gamma et al., 1995]:

```
...
if (instance not created yet) // read guard
  create instance
  mark instance as created // store in guard
end if

return instance
```

This snippet shows a pseudo assembler code that is generated from the method *device()* (first listing). As illustrated here, the guard variable is used to control the instance creation. Typically, the guards store the value 1 (*true*) to indicate that the instance has been created, and 0 (*false*) otherwise.

Destructor list entry for lazy initialisation. For each lazy initialised instance, the compiler generates the corresponding entry in the destructor list. This entry is a sort of a registration mechanism for ordered instance destruction.

Padding regions. Unused bytes may be inserted by the compiler to align the adjacent binary objects.

Virtual function table. This table is created for each class with virtual methods and is used at runtime as a dispatcher mechanism when calling virtual methods (it decides which class is responsible for execution).

3 Proposed Method

Realising mixed-criticality as proposed here starts with a software project provided in form of sources. Based on safety requirements that define how software and hardware resources shall be split up according to criticality, the responsible safety engineer/architect makes use of language annotations to select the related variables in the source code[1]. During the standard compilation process, all data artefacts (including also parts of the C/C++ runtime from Sec. 2.2) are identified in the runtime memory and classified based on annotations in code.

The result of this process is a list of memory regions enriched with criticality information. This list can then serve different purposes, for example to configure specific self-tests or memory protection measures or to provide a detailed documentation of how the system is organised with regard to criticality.

The foundation for identifying all data artefacts in runtime memory is the traceability analysis between the source code and memory, which we deploy as part of the standard compilation process. In the following, we describe the traceability analysis and the aforementioned language support for classifying data in a more detail.

[1] With ”related variables” is meant, data that belong to software or hardware resources of a specific criticality. Example: an input value, representing the state of an safety interlock switch; another example is the introduced *deviceManagerInstance* object.

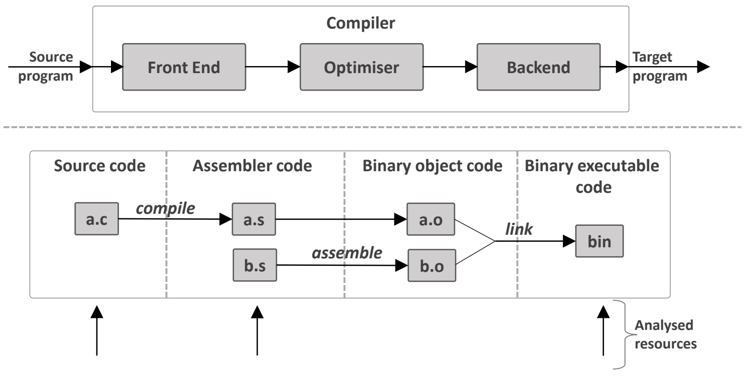


*Figure 2: Compiler pipeline (**top**) [Hathhorn, 2012], resources generated during the compilation process (**bottom**) [Rosenblatt, 1993]; arrows below show the resources that are processed in the course of the traceability analysis.*

3.1 Compiler-enforced Traceability Analysis

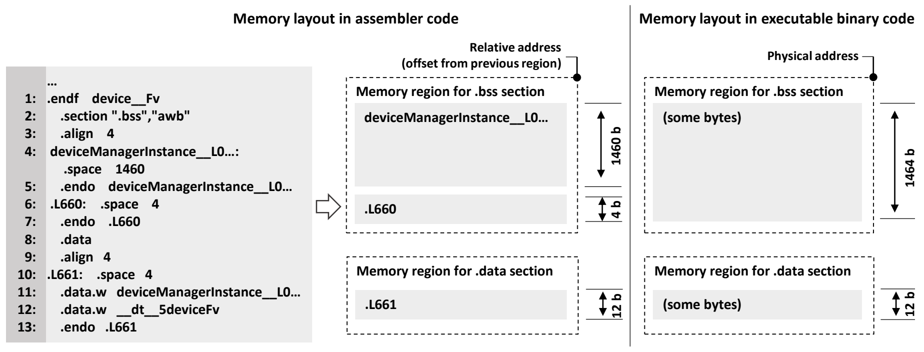
As already introduced, the memory layout extracted from binary object and executable files is incomplete. For example, when disassembling an ELF executable, one can localise global or file-scoped binary objects, i.e. those which have been defined in source files as global or static class member variables for example, but not local static variables nor parts of the C/C++ runtime. Even linker map files, which normally provide more details about memory, do not contain the complete information. Fortunately, different parts of the compiler pipeline generate fragments of the memory layout which, when combined, may help to recover the missing information. Thus, our method is based on the following premise:

- Physical view of memory (see Fig. 1), which can be read from binary or map files, provides a course grained memory organisation in which it is possible to observe information about (**a**) memory regions that belong to a file and (**b**) binary objects in global and file scope.
- Parts of the C/C++ runtime system can only be recovered in form of address offsets relative to binary objects identified in the physical view. These address offsets can be recovered from intermediate assembler files.
- Symbolic names of variable definitions and sequence of variable definitions in the source files provide a means to establish consistent mapping between the code and runtime memory.

For establishing traceability links based on the points above, we perform a thorough analysis of the binary and assembler code and map the identified binary objects to source code elements (variables/class instances), see Fig. 2. Some relevant aspects of this analysis are described in the sections below.

3.1.1 Source Code Analysis

For a given source code project (a set of C/C++ files), all definitions (variables/class instances) including their type and identification information are extracted. In order to



```
    ...
1:  .endf   device__Fv
2:    .section ".bss","awb"
3:    .align   4
4:   deviceManagerInstance__L0...:
      .space   1460
5:    .endo   deviceManagerInstance__L0...
6:   .L660:   .space   4
7:    .endo   .L660
8:    .data
9:    .align  4
10: .L661:   .space   4
11:   .data.w   deviceManagerInstance__L0...
12:   .data.w   __dt__5deviceFv
13:   .endo  .L661
```

Memory layout in assembler code
Relative address (offset from previous region)
Memory region for .bss section
deviceManagerInstance__L0...
1460 b
.L660
4 b
Memory region for .data section
.L661
12 b
Memory layout in executable binary code
Physical address
Memory region for .bss section
(some bytes)
1464 b
Memory region for .data section
(some bytes)
12 b

*Figure 3: Memory layout of the function device() in assembler code (**left**) and binary executable code (**right**). The illustration shows how missing information in the binary code is recovered by the use of the memory layout in the assembler code. The binary code provides physical locations of some memory regions, but not the internal structure of those memory regions and no traceability links to all source code and compiler generated data (indicated as some bytes).*

reliably identify these details from project sources, we use the C++ Development Tooling (CDT) [Piatov, 2012].

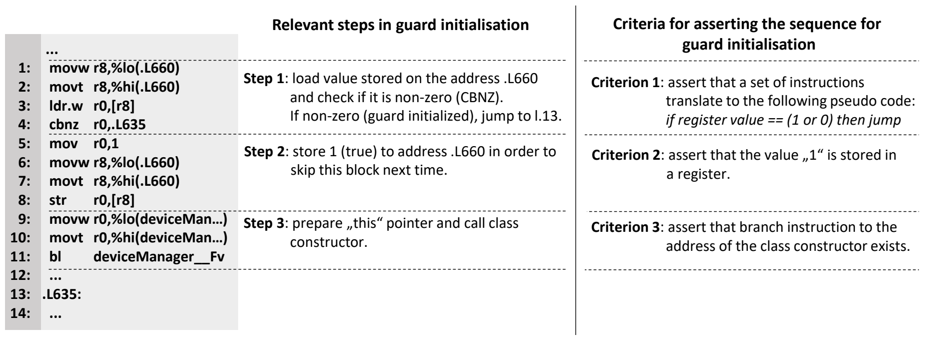
The output of the analysis is a list of definitions, in particular global variables, local static variables and static class member variables, enriched with type and identification information.

3.1.2 Memory Analysis

Identification of memory regions. The first step in recovering the memory layout is gathering the location of memory regions that belong to source files. An illustrative example is shown in Fig. 3, where the memory layout of our function *device()* is represented by two memory regions (right part in figure). As shown here, the two memory regions do not exactly specify the content inside, but only their physical location and size are known at this point.

Identification of binary objects. The concrete content of memory regions observed in the first step is identified by analysing assembler files. The assembler files do not provide any information about the overall memory layout of the project, but they provide detailed information about the individual binary objects. An example of such a detailed memory layout is given in Fig. 3 on the left. The lines 2-13 define the memory layout generated for the function *device()*. The assembler directives shown in this snippet instruct the compiler to generate the memory layout as shown in the middle: one region is created and allocated to *.bss* section, another one to *.data* section. Further, the first region is split up into two further regions: the one for the variable *deviceManagerInstance*, another one for its guard. The memory region in *.data* section is reserved for a destructor list entry (i.e. a registration entry for ordered destruction of an instance).

Classification of binary objects. The above part of the analysis just identifies binary objects of the C/C++ runtime system, but their type is not known yet. Assuming that there



```
     ...
 1:  movw r8,%lo(.L660)
 2:  movt  r8,%hi(.L660)
 3:  ldr.w  r0,[r8]
 4:  cbnz   r0,.L635
 5:  mov    r0,1
 6:  movw r8,%lo(.L660)
 7:  movt  r8,%hi(.L660)
 8:  str      r0,[r8]
 9:  movw r0,%lo(deviceMan…)
10:  movt  r0,%hi(deviceMan…)
11:  bl       deviceManager__Fv
12:  ...
13: .L635:
14:  ...
```

Relevant steps in guard initialisation	Criteria for asserting the sequence for guard initialisation
Step 1: load value stored on the address .L660 and check if it is non-zero (CBNZ). If non-zero (guard initialized), jump to l.13.	**Criterion 1**: assert that a set of instructions translate to the following pseudo code: *if register value == (1 or 0) then jump*
Step 2: store 1 (true) to address .L660 in order to skip this block next time.	**Criterion 2**: assert that the value „1" is stored in a register.
Step 3: prepare „this" pointer and call class constructor.	**Criterion 3**: assert that branch instruction to the address of the class constructor exists.

*Figure 4: Managing guards during creation of class instances: exemplary assembler code (Cortex M3, Thumb-2) (**left**); relevant steps that the code executes (**middle**); criteria used to check whether this piece of code corresponds to guard initialisation or not; note: in this example, if Criteria 1, 2 and 3 evaluate to true, the guard address can be read from the processor register R8 (**right**).*

is no explicit type information provided, classification of binary objects based on type is performed by simulating fragments of assembler files (cf. flow-based type inference and pattern-based variable recovery in [Caballero and Lin, 2016]). During simulation, after executing each line of assembler code, the content of processor registers is checked against the specified expected sequence in code. This expected sequence corresponds to the steps required to evaluate a guard (or a destructor list entry) as shown in the code snippet in Sec. 2.2. After the match has been found, the memory address of the guard or destructor list entry can be read out from one of the simulated processor registers.

Fig. 4 illustrates this: left part in figure shows a compiler generated fragment of the method *device()* in which the guard is managed. This code executes the aforementioned *if-then-else* statement from Sec. 2.2 as follows (middle part): check if the guard has already been initalised (lines: 1-4); if not, initialise the guard and create a class instance (lines: 5-11); else, skip the initialisation (lines: 13-14). Matching against the expected sequence is done by evaluating 3 criteria points, shown on right in figure. A piece of code can be considered as a guard initialisation sequence only if all criteria points are fulfilled. In a positive case, the guard memory location can be read from one of the processor registers – in this example code, the register *R8* is the one with the guard location *.L660*[2].

[2] Note that the example here is a very simple form of the guard management. The way how registers are used, initialised as well as whether or not one guard is reserved for one instance is not only compiler specific, but also depends on the used code optimisation strategy. For example, aggressive optimisations may remove "redundant" code and use one single guard for managing multiple class instances that belong to the same block in the code. It is also worth mentioning that the code in this example is fully expanded, i.e. without sub-routines. Some compilers may even simplify this behaviour in a way that only registers are prepared as shown on lines 1-3 in Fig. 4, and the rest is covered by runtime libraries. For example, variants of the *GNU GCC for ARM* compiler use *__cxa_guard_** library functions to manage guards. In multi-threaded system model, these functions are a bit more complex than the code from Fig. 4, since they also include specific thread synchronisation logic.

Mapping binary objects to physical address space. Finally, recovery of the memory layout in Fig. 3 (right) is completed after the mapping of memory regions between assembler code and binary code has been established. In our example code from Fig. 3, the first memory region in assembler code is directly mapped to the *.bss* memory region in the binary code. The second memory region in the assembler code is located at offset 0 from the previous memory region, so it is mapped to the *.bss* memory region at an offset of 1460 bytes. Last, the third memory region (destructor list entry *.L661*) is mapped to the address of *.data* section plus an offset of 0 bytes.

3.1.3 Source Code to Memory Mapping

After the memory layout in binary has completely been recovered and all binary objects have been classified, the source variables/class instances are mapped to memory.

Name-based mapping. The definitions (variables/class instances) are identified in memory by matching symbolic names between different data artefacts. Before mapping can be performed, all variables and binary objects get assigned qualified names which are unique in the project (e.g. *DeviceManager/device/deviceManagerInstance*).

Location-based mapping. For all other binary objects – e.g. when a guard has to be assigned to a variable – the order of variable definitions in methods is used for mapping. A good representative example that illustrates what this *order of definitions* means is given in the code snippet in Fig. 4. Typically, the guard that manages the class instance is accessed in a code block or scope within which the instance shall be created. In this example, the instance is created at lines 9-11, which belong to the same scope where the guard is evaluated (lines 1-4). Based on this, the guard can be considered as belonging to the class instance. The same applies for destructor list entries.

Mapping these parts of the C++ runtime system to code variables/instances is relevant in that it allows to enforce the classification of the C++ runtime system based on criticality. For example, if our *deviceManagerInstance* shall meet safety integrity IEC61508 SIL3/SC3, all parts of the C++ runtime system that depend on this data (i.e. guard, destructor list entries, virtual tables, ...) may need to be classified as such too[3]. The following section describes how this classification propagates from the source down to the the C++ runtime system.

3.2 Language Support for Defining Criticality

Assigning criticality to hardware and software resources is supported via annotations in project sources. The following listing illustrates use of annotations.

```
...
// this global instance is assigned Safety Integrity 1
DataAnalytics dataAnalyticsInstance_ IEC61508_SIL1;

//-----------------------------------------------------------
DeviceManager& device(void)
```

[3] Note that this is does not necessarily apply to all systems; it is rather a topic that needs to be clarified in the context of FMEA for a specific system. For example, if object destruction is of no avail because the system if fully static, destructor list entries cannot influence the system, thus they can be ignored in the analysis.

Data artefact	**# of data artefacts in project / size [byte]**			
	Project 1		**Project 2**	
	Safety	Normal	Safety	Normal
Variable/instance	100 (100) / 12215	52 (52) / 7070	208 (208) / 17081	194 (194) / 20616
Guard	47 (0) / 188	9 (0) / 36	124 (0) / 496	11 (0) / 44
Constructor list	1 (1*) / 32	-	1 (1*) / 24	-
Destructor list	1 (1*) / 8	-	1 (1*) / 16	-
Destr. l. (lazy)	1 (1*) / 8	-	1 (1*) / 8	-
Destr. l. entry (lazy)	-	43 (0) / 516	-	50 (0) / 600
Padding regions	-	8 (0) / 18	-	13 (0) / 23
Virtual fun. table	129 (129*) / 4616	-	155 (155*) / 5896	-
Improvement [%]	min. 3.08, max. 21.95		min. 2.60, max. 15.86	

Table 1: Memory layout generated for two projects (Project 1: ~100kLOC, Project 2:~150kLOC, GHS Compiler for ARM Cortex M3, ELF). ***Note 1****: values in parentheses represent the number of data artefacts which can be identified using standard compiler features such as memory sectioning for example.* ***Note 2****: * indicates that identification of these data artefacts is not always feasible.*

```
{
   // this instance is assigned Safety Integrity 3
   static DeviceManager deviceManagerInstance IEC61508_SIL3;

   return deviceManagerInstance;
}
```

For the sake of understanding, another object *dataAnalyticsInstance_* has been defined in addition to our example *deviceManagerInstance*. This object is required to meet different safety integrity than *deviceManagerInstance*, thus, different safety measures apply there. The annotations for assigning specific criticality levels are provided in form of C/C++ preprocessor macro definitions, illustrated here as *IEC61508_SIL**.

The way how these annotations enforce assigning criticality levels within the runtime memory layout can be summarised as follows: during compilation phase, the source code parser – CDT [Piatov, 2012] – recognises annotations and uses them to classify all source variables in the project. In the end of the compilation phase, these properties are then applied from sources variables to binary objects. Note that within the recovered memory layout at this compilation phase, as discussed in Sec. 3.1.2, binary objects are already mapped to both source variables and memory regions, thus, the annotations apply to all related memory regions.

4 Experimental Evaluation

As final words to our example, the following memory regions would be generated for the *deviceManagerInstance*: raw data (1460 bytes), guard *.L660* (4 bytes) and virtual destructor list entry *.L661* (12 bytes). Note that due to lazy initialisation, no entries in the global constructor and destructor lists are made here.

In this section, we evaluate the proposed method and summarise our findings by considering two industrial projects.

4.1 Application Use Case

The proposed method has been recently evaluated and applied during development of software for safety-critical controllers used in process and industrial automation. In a nutshell, these controllers are part of a safety instrumented system, and are responsible for providing sensor values to the logic solver (a PLC) and forwarding the values to the actuators.

The controllers are developed to meet SIL3/SC3 requirements of the IEC61508 standard and implement a number of safety measures to mitigate systematic and random hardware failures. The proposed method has been utilised to provide an accurate isolation between safety-related and non-safety-related software and these data are used to feed specific software-implemented self-test algorithms.

4.2 Performance Figures

The main objective behind this work was to classify each byte in memory and in this way to address missing information about the hidden data in memory. To give an insight into how many of these hidden data objects are generated by compilers, we analyse two software projects which have been conducted for the two functionally similar safety controllers, but with different complexity.

Tab. 1 provides an overview of the memory organisation for both projects. The first value in a cell shows a number of data artefacts identified by the proposed method – this is also the total number of data artefacts in a project; the number in parenthesis shows how many binary objects would be identified when using standard compiler features (e.g. memory sectioning); the last number shows the total memory consumption for the collected binary objects in bytes. For example, 155 virtual tables that in total occupy 5896 bytes have been identified in the Project 2. Using standard compiler features it may be possible to identify all of the 155 virtual tables, but this is not always guaranteed. Therefore, the improvement shown in the last row has been represented as a range.

The table also shows how we split up the data according to criticality, but the purpose is merely to demonstrate that also parts of the C++ runtime system can be classified individually with regard to their criticality. The essential point here is that significant part of memory can now be identified. As highlighted in the last row, applying the proposed method increases the number of identified bytes by 3.08% to 21.95% for Project 1 and 2.60% to 15.86% for Project 2. For safety-critical data, this improvement is as follows: 1.10%-28.43% and 2.11%-27.38% for Project 1 and 2 respectively. Note that these values strongly depend on the nature of the project, in particular on its complexity, and how software is organised with regard to criticality. For example, Project 2 has more safety-critical variables, more safety-critical guards and therefore higher min. improvement than Project 1.

4.3 Qualitative Summary

Another advantage of considering each byte in memory as proposed here is the ability to produce a detailed evidence about how the project is organised with regard to different criticality levels. This may serve as a solid basis for justifying the isolation between those criticality levels.

On the other side, the proposed method comes with a few limitations. First, as part of the standard compilation process, the instrumentation code adds some performance penalties. This may not be a problem when occasionally generating evidence documents, but may be of relevance when the instrumentation is used in each compilation campaign, for example when creating an input for self-tests.

Second, in a case where the instrumentation is used to produce an input for various self-tests, failures in the memory analysis need to be taken into account. For example, wrongly assigned criticality may lead to ineffectiveness of the corresponding self-test. In our setting, we performed an extensive tool qualification of our compiler extension according to IEC61508:2010 T3 requirements [4].

Last, part of the analysis where the assembler code is simulated is strongly depending on the used compiler and processor. Thus, maintaining such a code adds to the overall development overhead, especially when different configurations of compilers/processors need to be supported.

5 Discussion

The proposed method has been originally developed under assumptions that hold for typical bare-metal or simple RTOS-based controllers in which criticality can only be assigned to fragments of memory. Yet, since the method does not directly depend on processor architecture, but solely on used language, compiler and binary model that defines the runtime memory layout, it could potentially be of benefit in other architectures or for other purposes. In this section, we provide an overview of some alternative applications.

5.1 Industrial Safety Controllers

The assumed system model that we've introduced in Sec. 1.3 can be found in a number of other application fields, including automotive, avionics, and medical devices, just to name a few. AUTOSAR Classic platform for instance [Staron and Durisic, 2017], which is a standard framework and middleware for developing software for Electronic Control Units (ECU) in cars, is designed to support diverse ECU architectures, ranging from compact low-cost controllers for monitoring tire pressure to high-performance and safety-critical systems for complex powertrain functions [Nilsson et al., 2008]. Although this middleware has a complex architecture that includes layers (i.e. separation of basic services, RTOS and applications), application software components and ”runnables” or tasks, all these building artefacts can be abstracted to primitive language elements, such as classes, methods and variables. Thus, using such a simplified abstraction, designing mixed-criticality in AUTOSAR based systems could be conducted as described in this paper. In the same way, other middleware solutions such as IMA (Integrated Modular Avionics) or DECOS (Dependable Embedded COmponents and Systems) [Schlager et al., 2006] could be addressed.

The above apparently holds as long as no advanced hardware- and OS-enforced partitioning scheme are used where criticality is assigned to the complete OS and applications and where the individual partitions run in virtualised environments or as OS processes (cf. LynxOS-178C [Leiner et al., 2007]). For these architectures, no benefits would be gained when applying the proposed method, because here the complete software stack including the underlying language runtime belongs to the same criticality. This, in the end means, that these architectures do not have the problem when they ignore to consider

[4] Note that the tool FMEA and tool qualification are out of scope of this paper and are therefore not discussed in detail here.

the language runtime when assigning criticality to software components. The language runtime is simply part of the same partition as its related software component.

5.2 Supporting Safety Certification by Evidence

Safety standards, such as ISO26262 and IEC61508, allow components of different criticality levels to be combined together in the same application, as long as they are free from interference. Freedom from interference, both in temporal and spatial domain, as discussed, is achieved by deploying mechanisms which guarantee that higher-criticality functions always get computing resources and that their state cannot be altered by lower-criticality functions. Ensuring spatial isolation is a more challenging aspect, since it requires built-in hardware features, such as Memory Protection Unit (MPU), Memory Management Unit (MMU) or virtualisation for example. If none of these mechanisms are available — which may be the case for some simple microcontrollers – it may be required to analytically demonstrate freedom from interference.

The IEC61508 standard, for instance, recommends an object code analysis as a means to perform this. In the scope of this analysis, it must be demonstrated that lower-criticality components cannot alter the state/data of higher-criticality components. The proposed method can be used here to provide a detailed location of all object in memory. It would, however, be necessary to extend the method for analysing memory accesses between those objects.

5.3 Supporting Fine-Grained Spatial Memory Isolation

Memory accesses in many modern microcontrollers are controlled by means of the MPU. This hardware feature allows sectioning memory into regions and assigning different access permissions to each of them. MPUs are commonly used in embedded devices to prevent bugs and malware from affecting critical processes. The main concern here is that the number of regions they can support is limited, and, apart from that, location and length of each of the regions is either fixed or can be selected out of a fixed set of available options. For example, Cortex M3/M4 cores support up to 8 memory regions, where the length for each of the regions can be selected from a set of values ranging from 32B to 4GiB [Yiu, 2013].

In consequence to this, partitioning of memory according to different criticality levels, for example in order to enable spatial memory isolation, must be carried out based on fixed memory layout the MPUs offer, which largely limits the flexibility in implementing mixed-criticality. This limitation holds for all current systems that use the MPU as a means to enforce spatial memory isolation.

The method we introduced here, apart from identifying objects in memory, also provides criticality information for each byte in memory and does not set any constraints on how memory is organised. Note that our method is meant to be used by software-implemented safety measures, therefore, the organisation of memory is not an issue for those measures. In order to utilise this method in implementing fine-grained memory protection, specific hardware support would be needed. In short, each byte (or a word – depending on desired granularity level) in memory would need a ”criticality flag”. When access to a memory byte or word is made, the processor would then be able to decide whether or not to prevent this access by issuing an interrupt, in the same way the MPUs do. This decision would be made based on a value of the criticality flag, and the criticality of the code that makes the request. This could be, for instance, the current mode of processor execution (privileged, user) or a program counter value in code memory. In the latter case, the criticality flag of the program counter could just be read out during evaluation of the access.

Realisation of such a memory access control could for example be made on Field Programmable Gate Arrays (FPGA). These hardware features, combined with the proposed method, would allow to implement a flexible and fine-grained spatial memory isolation for simple (bare-metal) microcontrollers.

6 Related Work

In this section, we review some closely related work. Topics of interest include (**1**) recovery of information from binaries, (**2**) memory sectioning and (**3**) source code to object code (binary code) mapping and traceability analysis.

Binary Analysis. Recovering fragments of code and data from binaries has been practised since the invention of compilers, yet it is today still an active research topic. Some of the contributions in this field that are closely related to our method can be found in [Urueña, 2008, Katz et al., 2018, Erinfolami and Prakash, 2019, Yoo and Barua, 2014, Fokin et al., 2011]. Although they address different goals in analysing binaries such as reuse of code, hooking and vulnerability detection [Caballero and Lin, 2016], and among other aspects target different controller architectures, there are many comparable features to what we have introduced here. For example, the work in [Erinfolami and Prakash, 2019] deals with protecting the routing through the class hierarchy by identifying virtual table objects in memory and inserting a special instrumentation code. Here, a more thorough analysis is performed as in our case, since the information is extracted from pure binaries (i.e. no source or intermediate assembler code available). In the course of the memory analysis, we are identifying the individual variables or class instances, but not all class internals, i.e. non-static class member variables. The work in [Katz et al., 2018] goes into details with considering the composite structures.

The methods proposed in [Yoo and Barua, 2014] and [Fokin et al., 2011] are very closely related to our analysis with regard to objects in memory they address. Except of guards and destructor list entries, they focus on identifying all objects of the C++ memory model, including features which are by default not used in safety development, such as RTTI and exception handling for example.

Memory Sectioning. Allocation based sectioning is a commonly applied method for identifying and classifying data in runtime memory layouts, where inherent compiler directives are used to position data in specified memory sections [Urueña, 2008], [Autosar, 2017]. Traditionally, sectioning is made by shaping the linker definition file, which specifies the final memory layout. According to the work in [Urueña, 2008], this file is generated out of special source code annotations in ADA on-board avionics software. Alternatively, the memory layout can also be controlled via standard compiler directives in the code or by instructing the linker directly [Autosar, 2017].

Source Code to Object Code Traceability Analysis. Ensuring traceability between the source and object code in context of functional safety is motivated by the fact that both representations of software may differ, since compilers may put additional branches in code, generate parts of the language runtime or link additional libraries. Although this has many other implications, traceability analysis is considered merely as a measure to demonstrate that code coverage criteria for module (unit) testing is met [Brauer et al., 2015].

Summary. Identification of all data artefacts of the C++ runtime system, their classification and assignment to the project (source code) with the purpose of improving the coverage of safety measures and quality of documentation (evidence for certification) under consideration of different levels of criticality has, to our knowledge, so far not been covered by the state-of-the-art.

7 Conclusion

In this paper, we have addressed a specific problem that affects some system architectures when dealing with mixed-criticality. In particular, when a system is to be organised with regard to different criticality levels, it must be ensured that criticality can reliably be assigned to all relevant resources (software components, hardware resources such as fragments of memory, etc.). This appears to be a problem for systems that run software with complex language runtime environments, such as the one of the C++ language. Parts of this language runtime are interconnected with the rest of software in a way that any fine-grained assignment of criticality to different software components cannot completely be conducted. This is due to the fact that compilers do not consider all objects in memory when code and data need to be allocated to specific memory segments.

In consequence, assigning criticality just by allocating pieces of code and data will result in incomplete spatial partitioning, i.e. parts of the language runtime may not be allocated as desired. Leaving out those parts may have an impact on functional safety, because, as we pointed out, correct functioning of higher-criticality software components may in some situations depend on the integrity of code and data that belong to the language runtime.

We have shown that with detailed source to object code traceability analysis, these fragments of memory can be recovered and mapped to the corresponding software components or variables in the source code, enabling in the end the classification of each object based on the required criticality level.

Our method works as an add-on to the standard compilation process and in this way comes, as any workaround, with limitations that we've outlined. We do not expect that such a workaround can serve as a long-term solution, it is rather something that should be provided as a standard feature of modern compilers.

Acknowledgements

The authors would like to express their gratitude to the guest editor, Dr Georg Macher, for his scientific support and collaborations in the process of writing the paper.

References

[Autosar, 2017] Autosar (2017). Memory Mapping.

[Brauer et al., 2015] Brauer, J., Dahlweid, M., Pankrath, T., and Peleska, J. (2015). Source-code-to-object-code traceability analysis for avionics software: Don't trust your compiler. SAFECOMP 2015, Berlin, Heidelberg. Springer-Verlag.

[Broy, 2018] Broy, M. (2018). The leading role of software and systems architecture in the age of digitization. pages 1–23, Cham. Springer International Publishing.

[Burns and Davis, 2017] Burns, A. and Davis, R. I. (2017). A survey of research into mixed criticality systems. volume 50, New York, NY, USA. Association for Computing Machinery.

[Caballero and Lin, 2016] Caballero, J. and Lin, Z. (2016). Type inference on executables. volume 48, pages 65:1–65:35, New York, NY, USA. ACM.

[Ebert and Favaro, 2017] Ebert, C. and Favaro, J. (2017). Automotive software. volume 34, pages 33–39, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Ebert and Jones, 2009] Ebert, C. and Jones, C. (2009). Embedded software: Facts, figures, and future. volume 42, pages 42–52.

[Erinfolami and Prakash, 2019] Erinfolami, R. A. and Prakash, A. (2019). DeClassifier: Class-Inheritance Inference Engine for Optimized C++ Binaries.

[Esper et al., 2015] Esper, A., Nelissen, G., Nélis, V., and Tovar, E. (2015). How realistic is the mixed-criticality real-time system model? RTNS ’15, page 139–148, New York, NY, USA. Association for Computing Machinery.

[Fokin et al., 2011] Fokin, A., Derevenetc, E., Chernov, A., and Troshina, K. (2011). Smartdec: Approaching c++ decompilation. pages 347–356.

[Gamma et al., 1995] Gamma et al. (1995). Design patterns: Elements of reusable object-oriented software. USA. Addison-Wesley Longman Publishing Co., Inc.

[Hardy, 2010] Hardy, T. (2010). The system safety skeptic: Lessons learned in safety management and engineering. AuthorHouse.

[Hathhorn, 2012] Hathhorn, C. (2012). Engineering a compiler, 2nd ed. volume 37, pages 36–37, New York, NY, USA. ACM.

[Kajtazović et al., 2020] Kajtazović, N., Hödl, P., and Macher, G. (2020). Instrumenting compiler pipeline to synthesise traceable runtime memory layouts in mixed-critical applications. pages 73–78.

[Katz et al., 2018] Katz, O., Rinetzky, N., and Yahav, E. (2018). Statistical reconstruction of class hierarchies in binaries. ASPLOS ’18, pages 363–376, New York, NY, USA. ACM.

[Leiner et al., 2007] Leiner, B. et al. (2007). A comparison of partitioning operating systems for integrated systems. pages 342–355, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Lippman, 1996] Lippman, S. B. (1996). Inside the c++ object model. Redwood City, CA, USA. Addison Wesley Longman Publishing Co., Inc.

[Macher et al., 2015] Macher, G., Sporer, H., Berlach, R., Armengaud, E., and Kreiner, C. (2015). Sahara: A security-aware hazard and risk analysis method. pages 621–624.

[Mössinger, 2010] Mössinger, J. (2010). Software in automotive systems. volume 27, pages 92–94.

[Nair et al., 1978] Nair, Thatte, and Abraham (1978). Efficient algorithms for testing semiconductor random-access memories. volume C-27, pages 572–576.

[Nilsson et al., 2008] Nilsson, D. K., Phung, P. H., and Larson, U. E. (2008). Vehicle ecu classification based on safety-security characteristics. pages 1–7.

[Piatov, 2012] Piatov, D. (2012). Using the eclipse c/c++ development tooling as a robust, fully functional, actively maintained, open source c++ parser. Berlin, Heidelberg. Springer Berlin Heidelberg.

[Rosenblatt, 1993] Rosenblatt, B. (1993). Learning the korn shell. Sebastopol, CA, USA. O’Reilly & Associates, Inc.

[Schlager et al., 2006] Schlager, M. et al. (2006). Encapsulating application subsystems using the decos core os. pages 386–397, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Simó et al., 2021] Simó, J., Balbastre, P., Blanes, J. F., Poza-Luján, J.-L., and Guasque, A. (2021). The role of mixed criticality technology in industry 4.0. volume 10.

[Staron and Durisic, 2017] Staron, M. and Durisic, D. (2017). Autosar standard. pages 81–116, Cham. Springer International Publishing.

[Traub et al., 2018] Traub, M., Vögel, H. J., Sax, E., Streichert, T., and Härri, J. (2018). Digitalization in automotive and industrial systems. pages 1203–1204.

[Urueña, 2008] Urueña, S. (2008). A new approach to memory partitioning in on-board spacecraft software. pages 1–14, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Yiu, 2013] Yiu, J. (2013). The definitive guide to arm cortex-m3 and cortex-m4 processors, third edition. USA. Newnes.

[Yoo and Barua, 2014] Yoo, K. and Barua, R. (2014). Recovery of object oriented features from c++ binaries. pages 231–238, Washington, DC, USA. IEEE Computer Society.