Static and Verifiable Memory Partitioning for Safety-Critical Systems

Jean Guyomarc'H, Jean-Baptiste Hervé

Static and Verifiable Memory Partitioning for Safety-Critical Systems

Jean Guyomarc'h
Krono-Safe
Université Paris-Saclay / CNRS / SATIE
Massy, France
jean.guyomarch@krono-safe.com

Jean-Baptiste Hervé
Krono-Safe
Massy, France
jean-baptiste.herve@krono-safe.com

***Abstract*—Multitasking enables multiple tasks to be executed on the same hardware, and spatial partitioning aims at enforcing a strong isolation between them: tasks must not access memory regions for which they were not granted permission. This behavior is enforced at run-time by memory protection schemes enabled by dedicated hardware components. Today, memory protection is widely implemented on a great diversity of systems, mostly with dynamic requirements (*e.g.* variable number of tasks). Safety-critical systems must comply with high level of certification to ensure minimal probability of failure and are subject to stringent requirements on the embedded executable, which makes memory protection mandatory, but requires important certification efforts. This paper presents a method for the generation of static and verifiable memory partitioning schemes towards safety-critical systems, aiming at reducing certification costs without compromising safety properties.**

***Index Terms*—static memory protection, spatial partitioning, certification of safety-critical systems**

I. Introduction

Multitasking enables for different tasks to share the same hardware execution context. It allows a modularization into independent entities that may communicate with each other, effectively allowing a distributed development. However, without spatial isolation, this approach incurs security and safety issues, as tasks would be free to read, write and execute data and code of other tasks. Spatial partitioning is a technique contributing to *fault containment*: tasks then can only access memory addresses which permissions were explicitly granted, effectively reducing the impact of a task failure on the overall system. As such, memory partitioning can be found in industrial domains with safety considerations such as avionics [1] or automotive [2].

Because of the hazards that may be caused by the failure of safety-critical systems, the process of developing these systems is complex and costly, with a relatively high time-to-market [3]. To minimize the risk of failure, static and immutable approaches are preferred to dynamic behaviors, because they can be analyzed more in-depth and verified off-line. Spatial partitioning is widely used today, and has received much attention for dynamic systems spawning tasks on-the-fly. However, purely static approaches that are well-suited towards safety-critical systems seem to have received less interest.

This paper contributes to reduce certification efforts of safety-critical systems, by presenting an automated and portable method enabling static and verifiable spatial partitioning schemes through configurable *memory positioning* and the generation of precomputed *memory protection tables* to be embedded in the certified binary. The generated artifacts would be subject to a validation process, which can be automated by appropriate tooling, in conformance with certification documents.

After detailing in Sec. II the background context and reviewing related work, we present in Sec. III a method performing off-line spatial partitioning. We then describe in Sec. IV an implementation of this method applied to an industrial use-case leveraging the NXP QorIQ P2020 platform before we conclude in Sec. V.

II. Background and Related Work

Spatial partitioning has been extensively documented in the literature for more than forty years to isolate tasks and ensure they do not perform illegal inter-task accesses [4]. It is a by-default option proposed by most modern operating systems over a wide range of devices, as long as they have built-in hardware support. Because this paper focuses on static approaches with a fixed set of tasks, systems with dynamic requirements are out of the scope of this paper, but for completeness, a review is presented by Achermann in [5]. We assume the execution of tasks to be controlled by a privileged *kernel*.

Hardware-assisted memory partitioning is mostly implemented by *Memory Protection Units* (MPU) or *Memory Management Units* (MMU). Assuming a proper configuration by software, their ability to raise hardware exceptions when invalid memory accesses are detected at run-time enables them to enforce security and safety requirements [1]. MMUs are usually more complex than MPUs because they aim at providing more elaborate functions such as virtual addressing and the notion of *memory pages*. Our work can be applied to both components, regardless of the memory model (flat or with virtual addressing).

Brygier et al. [6] state that the PikeOS microkernel configures the MMU statically at boot-time in a way that it cannot be re-configured during the nominal execution of their system. It is not explained whether the memory protection configuration tables are computed on-line during boot or if

they were precomputed (in that case, the process of generation is undocumented). Similarly, Perret et al. [7] also claim that their system makes use of a static MMU configuration at boot-time only. But they also do not explain how this configuration is generated.

The closest work to ours seems to be described in papers from Camier et al. [8] and Louise et al. [9], who mention that spatial partitioning of their real-time kernel OASIS is performed by loading MMU configuration tables that are statically generated at compile-time. However, the generation process is not explained. Similarly, David et al. [10] claim to have invented a system making use of similar tables, but without providing any detail on their generation.

To the best of our knowledge, no method aiming at automatically performing precomputed, static and verifiable spatial positioning for safety-critical systems has been documented in the literature.

III. Method

We present in this section an automated, compiler-agnostic and portable method to generate certifiable executables with static spatial partitioning that embed their own precomputed memory protection configuration as verifiable binary data.

A. Leveraging qualified toolchains

Toolchains are mostly composed of a *compiler* that converts a source file (probably written using the C programming language) into binary object files, and a *linker* that generates a single binary executable from input object files and a *linker script* that describes the final layout of the executable. Safety-critical systems usually rely on *qualified* toolchains, that have been chosen by industrials in the certification process and cannot be substituted with third-party components. As such, the method we propose uses exclusively unmodified off-the-shelf toolchains for compilation and link operations. The following assumptions on the toolchains are made in this paper:

1) the linker must be configurable through the use of *linker scripts*, or an equivalent feature. In the rest of this paper, we assume linker scripts are used to control the execution of the linker.
2) The compiler and linker must be independent programs that can be called separately. Although this is not a strict requirement, it makes the method easier to describe and probably simplifies the industrial process as well, by avoiding the propagation of source code throughout different entities.
3) The linker is *deterministic*: it always produces the same output for a fixed and ordered set of inputs.

B. Identify hardware requirements

Because memory protection is implemented by diverse hardware components, they may induce various additional constraints on memory partitioning. These usually consist in alignment requirements: a range of memory addresses to be protected with specific permissions may be aligned with respect to hardware-dependent requirements. The following examples illustrate this diversity:

- the MPU on single-CPU Cortex-M3 requires protected addresses to be aligned to a multiple of the size of the protected region;
- the MPU of the MPC5777M microcontroller by NXP allows arbitrary range size support without alignment constraints;
- the MMU of the T1042 and P2020 platforms by NXP allows to protect pages aligned on powers of four kilobytes (the size of a page must be 4^n kB with $n \in [1, 11]$).

The identified hardware constraints (*e.g.* alignment, MPU/MMU, page levels) have a direct influence on the next steps of the method that are detailed in the next sections.

C. Memory positioning configuration

Memory positioning consists in placing objects of interest at specific memory addresses, effectively enabling to group together objects to be protected with the same permissions. It may also be used to take full benefit of the hardware capabilities, such as core-local memories, but this side-effect of memory positioning is not detailed as it is out of the scope of this paper.

Linker scripts enable to perform memory positioning by instructing the linker, during the link operation on how *input sections* belonging to input object files are to be integrated to *output sections* in the final binary. Positioning may be explicitly performed by the application designer by specifying where objects must be placed; but because this is a tedious process, it is more convenient to consider a higher level of abstraction.

In this section, we briefly present an abstracted view of memory positioning tied with memory protection requirements, which together enable a concise descriptive view of spatial partitioning.

1) Groups: it is often needed to aggregate multiple input sections in one output section. For example, placing all input sections containing read-only constants of a set of objects in one output section. Every object involved in the application shall belong to one or more groups, effectively describing positioning relations between input sections and output sections.

2) Domains: as hardware memory descriptors may be in small amount (especially when an MPU is used), it may be convenient to group together output sections with the same memory permissions. As such, a domain is a collection of adjacent output sections that share the same memory permissions.

3) Regions: represent the available hardware memories, and as such describe a fixed-size range of physical addresses. They specify the final memory layout by containing the *domains* presented earlier.

These concepts are illustrated in Fig. 1, where *groups* are shown by the mapping of input sections to output sections in the final binary. Three execution contexts are presented: kernel, task A and task B. In this example, the kernel may not access any running task, so all the sections related to task

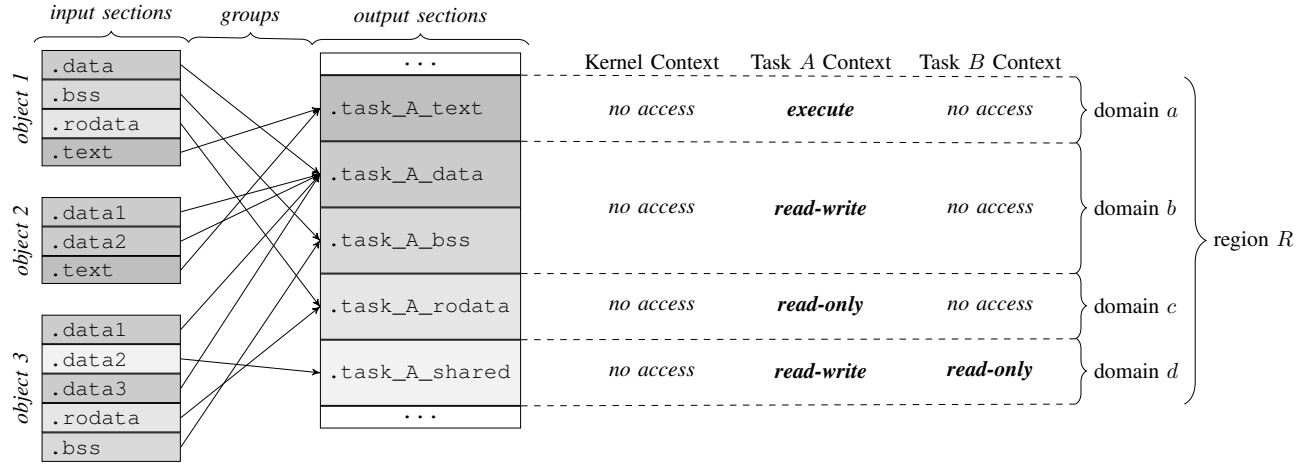

Fig. 1. Illustration of memory positioning where the input sections of three objects used by the task A are mapped to the output sections constituting the final binary. The arrows indicate the positioning strategy specified by *groups*. For example, the section `.data2` of *object 3* is to be contained within the section `.task_A_shared` of the binary. On the right of output sections are shown examples of different memory permissions to be applied depending on which context is running, described by four domains (a, b, c and d). For example, the output section `.task_A_shared` can be accessed in read-write when task A is running, read-only when task B is running, and cannot be accessed at all by the kernel. All these domains are placed in one physical memory region R.

A are tagged as *no access* when the kernel runs. Similarly, task B shall not access data or code owned by the task A, at the exception of a shared data area that task B may only read. When task A runs, the traditional permissions model is applied on its own code and data. It can be seen that several output sections are covered by one domain (domain b) when they are to be protected with the same permissions. This allows to use less memory descriptors at run-time, saving performance.

D. Automated binary generation process

The ultimate goal of the method is to generate an object file containing memory protection tables providing ranges of addresses to be protected. For obvious security and safety reasons, these tables must contain their own memory protection configuration, so the memory protection policy cannot be altered at run-time. These can be referred to from other object files through the definition of a *symbol*. This enables the executable to embed these information with a simple link operation, without requiring post-link modifications, leveraging the use of qualified linkers. However, as explained later, generating such tables may be challenging.

Because this method relies on an off-the-shelf linker, the link operation is considered atomic: addresses and sizes are only known at the end of the link operation. Because the linker may perform link-time optimizations, such as *relaxation*, which may affect the size of output sections, the sizes cannot be reliably determined before linking has completed. This is especially problematic to describe memory pages of variable size: depending on the hardware, they may be required to be aligned on their size, which is unknown until the link operation has finished. This makes it impossible to specify proper alignment constraints to the linker. Assuming that all pages have the same size avoids this problem, but forces suboptimal memory paging. In the general case, this causes a *cyclic dependency* between the object defining the protection tables and the binary that contains them, because the tables require addresses and sizes, and memory positioning is constrained by sizes, which are only known after the link has completed.

Purely static positioning (*i.e.* specifying physical addresses in the linker script) would remove this dependency, because addresses and size bounds would be known before the link. However, it requires manual intervention and a possibly tedious trial and error process in case of link failures. The method we propose manages to automatically and systematically break this cycle to generate a binary containing exact self-describing tables.

Fig. 2 illustrates the overall method, which is iterative and convergent. A *generator* is a program to be written depending on hardware constraints and the syntax of the linker script. It processes memory positioning configuration and, *if available*, the binary generated by a previous iteration. Of course, the binary is unavailable for the first iteration. Then, it generates a linker script and an object file by calling a compiler from the toolchain (for example the C compiler) on a generated source file. These generated files are then used as-is by the linker so an executable binary can be generated. When the embedding of self-describing tables is complete, the binary shall be ready for validation, otherwise a new iteration must be performed. Trivial requirements may allow to greatly simplify the generation process, we focus here on a more general case.

Before the first link can happen, protection tables must be generated, so the linker can find the symbol pointing to them. Otherwise, the symbol would be undefined, and the link would fail. To allow the first link to complete, a *dummy* symbol must be generated. It would point to meaningless data, resulting on an binary that would malfunction if executed. This binary would be taken as an input of the next iteration. At the $n+1$ iteration, the generator is able to extract addresses and

sizes from the binary produced at iteration n, by using binary analyzers such as LIEF [11]. These additional information provide the generator with more context to instantiate protection tables that would be linked during this iteration. The process ends when the data structures produced by the generator hold every address and sizes of domains to be protected, included themselves.

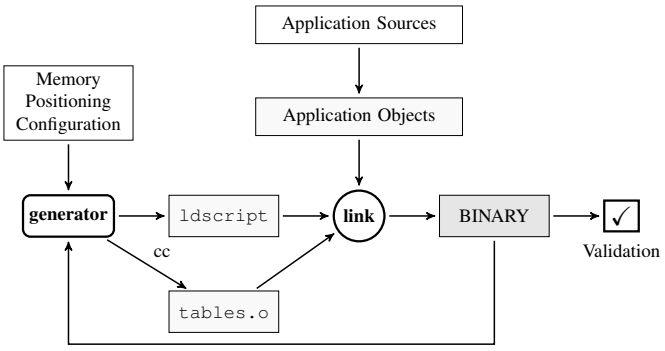

Fig. 2. Compiler-agnostic, hardware-specific method enabling the off-line generation of a verifiable static memory configuration tables embedded within the application binary. A cycle of link operations involving application objects, a generated linker script and a generated object containing protection tables ensures the generation of the binary with self-describing protection tables.

E. Verification strategy

Memory protection tables generated by the method presented in Fig. 2 can be seen as *Parameter Data Items* (PDI) considered by the DO-178C airborne certification standard: configuration data that influence the run-time behavior of the system without modifying the executable object code [12]. Binary analyzers (*e.g.* ELF decoders) can retrieve information from this table for validation purposes. These are used by qualified tools, such as the (in-development) suite *ASTERIOS Checker* described by Methni et al. [13] that verify the correctness and the compliance of the PDI embedded in the binary with the specification. This implies that the generator itself does not need to be qualified, because its outputs are verified with a qualified tool.

The correctness of the memory positioning can be estimated by verifying that the resulting memory layout matches the specifications as described by the positioning abstraction presented in Sec. III-C. Interestingly, this abstraction enables to factorize validation algorithms; if portions are of course target-specific, most work is actually target-agnostic. It effectively simplifies the development effort of the checking tool.

The main benefit of this approach is to greatly reduce certification efforts. Data that are generated off-line are less costly to verify than developing run-time code that must be fully-compliant with respect to certification processes. Also, core parts of the checker tool can be re-used among a variety of projects, reducing even more development efforts.

F. Portability

As previously stated, the method must be implemented for a given hardware and a specific compiler. Its implementation can then be reused on an arbitrary number of applications that are built with this compiler and this hardware. This portability property among applications allows significant reuse between industrial projects that share the same hardware and compiler base, including the validation strategy presented earlier.

To illustrate the genericity of the presented method, it has been implemented at Krono-Safe for various hardware platforms, including MPPA Coolidge by Kalray, MPC5777M by NXP, QorIQ by NXP (P2020, T1042), Aurix TriCore TC-39x by Infineon or STM32MP157 by ST. It supports several off-the shelf compilers such as GCC, the Wind River Diab Compiler or the Tasking compiler by Altium.

IV. Use case: spatial partitioning on NXP P2020

In this section, we study one industrial use-case at Krono-Safe: generating off-line static and verifiable tables describing the memory protection of a safety-critical avionic application on the NXP QorIQ P2020 platform. This application relies on the ASTERIOS real-time kernel and its associated toolchain, which are described in [13]. We detail how the method presented in Sec. III has been implemented to generate a certifiable executable embedding self-describing memory configuration tables that can be validated off-line.

A. Toolchain

The Wind River Diab Compiler is the toolchain involved in this use-case. The application sources are written in C and compiled to object files by a C compiler named `dcc`. The link operation, which can be driven by linker scripts, is performed by the linker `dld`. The name of these programs is used later in this section to refer to portions of the toolchain.

B. Hardware overview of the NXP QorIQ P2020 platform

The NXP QorIQ P2020 platform is based on e500v2 PowerPC cores by NXP [14]. On this hardware target, memory protection is implemented by a MMU through the software configuration of Translation Lookaside Buffers (TLB). Each core has two TLB arrays (named TLB0 and TLB1), with two cache levels each. The level one caches are entirely hardware-managed, and as such are out of the scope of this paper because they cannot be directly controlled through software. Level two caches are unified, meaning that they contain both data and instruction pages. If the page size is fixed to 4 kB for TLB0, they are of variable size for TLB1 (from 4 kB to 4 GB). It is required that pages must be aligned on their size.

MMU pages can be loaded into the TLB arrays through a set of *MMU Assist Registers* (or `MAS`) that must be set to appropriate values. After the `MAS` registers are configured, calling the TLB write entry instruction (`tlbwe`) makes the page load effective. Upon context switches, TLB entries are evicted by triggering a *Flash Invalidate* of the TLB arrays by writing a bit in a separate MMU control register. Entries are used by different execution contexts, such as userland and kernel. Userland is used to run user tasks with few privileges. The kernel, which is responsible of changing memory protection upon context switches, runs with supervisor privileges.

Kernel entries must be loaded at any time because in case of an interruption or a system call, the code will branch directly in the exception vectors, without a chance to load other descriptors, effectively preventing the system from triggering kernel code. As such, these entries are loaded to the TLB1 array with a special bit set: `IPROT` (invalidation protected), to prevent their invalidation.

C. Identification of run-time data

As the TLB arrays are rather different (fixed versus variable page size, set/way and fully associative, number of available entries), the usage that the kernel has of each TLB arrays are different.

1) Run-time data required by TLB0: as the TLB0 array has a fixed page size of 4 kB, and a large number of entries (512), one memory domain will be covered by several 4 kB pages. So, we need to generate several page descriptors per domain. As it is set/way associative, we need to know the final address of each page to know in which set it will be loaded by the TLB array. Only then, we can compute the index of the way to be used for each page, knowing the pages that would already be loaded when we load the new one. The information needed by the operating system to load the appropriate page descriptors upon context switches are to be stored as pre-computed, read-only data structures:

- the start address of the domain;
- MAS2 indicators such as write-through, cache-inhibited;
- MAS3 permission attributes;
- the number of 4 kB pages to create;
- for each page to be created, the *Entry Select value*, which is the index of the way in the set.

2) Run-time data required by TLB1: as the TLB1 array has fewer available entries, but supports variable page size, one domain can be protected by one page. We will change the size of the page to match the size of the domain. The following run-time information are required by the operating system:

- the start address of the domain;
- MAS2 indicators such as write-through, cache-inhibited;
- MAS3 permission attributes;
- the $TSIZE$ value, which is given by the size of the protected domain. The final page size is computed as 4^{TSIZE} kB.
- The *Entry Select value*, which is the index of the entry within the TLB array.
- The value of the `IPROT` flag, to prevent eviction of entries.

D. Generating the spatial descriptors

The final spatial descriptors are expected to contain the exact addresses, size and entry select values of all the memory domains, including themselves. It is indeed mandatory that memory protection cannot be overridden at run-time by rogue code execution. Because the TLB arrays may have different page sizes and because they have to be aligned on their size, off-line generation of spatial descriptors is laborious. Some information, such as size boundary addresses are unknown until a link operation.

A possible implementation of the generic method proposed in Sec. III would be to perform a chain of four link operations. This is illustrated in Fig. 3, which shows the four different steps, that are detailed below.

1) Determining the size of application domains: Before a link can occur, protection tables must be generated, because some object files refer to the symbol indicating where protection tables are defined. But because these cannot be fully generated until a link has been completed, *empty* tables are instead generated (in the object `empty.o`). This effectively allows the required symbol to be defined, enabling the initial link to complete. The resulting binary is of course ill-formed and cannot be used as-is, because protection tables are meaningless. However sizes of output sections (and therefore of domains) can be deduced from an analysis of the binary. At the end of this first link, the following information can be deduced from a binary analysis:

- the size of each domain covered by the TLB0;
- the number of TLB0 entries;
- the size of each domain covered by the TLB1 (with the exception of protection tables) ; and
- the number of TLB1 entries.

2) Determining the size of all domains: Thanks to the first link, the exact count of descriptors for TLB0 and TLB1 are known. This enables to generate *hollow* protection tables that are properly sized (in the object `hollow.o`). Their contents is still meaningless, but the linker will allocate the required amount of memory to hold the final protection tables. The resulting binary is still ill-formed, but it can be inspected to retrieve the exact size of every domain.

3) Determining the address of all domains: Because the result of the second link gives all the sizes, it is now possible to generate a linker script that align domains on their own size, which enables to freeze the spatial partitioning of the domains. The protection tables are still *hollow* at this point, because the addresses of domains are still unknown. The resulting binary enables to retrieve the exact addresses of every domain.

4) Generating the final binary: The result of the third link enables to populate protection tables with meaningful values because all the addresses and sizes are known. The fourth link is performed by replacing the hollow tables with the final tables. Since the final tables are placed at the same address than the hollow tables and because they have the same size, the other domains shall remain untouched. A final inspection on the resulting binary allows to verify that this assertion holds. The final binary is then processed by ASTERIOS Checker (at least partially, because this tool is under development) to verify its memory layout respects the specifications of the system.

V. Conclusion

We have presented a compiler-agnostic method contributing to the spatial partitioning of certifiable executables, which has been developed at Krono-Safe and implemented on a variety of hardware platforms. It features the generation of precomputed

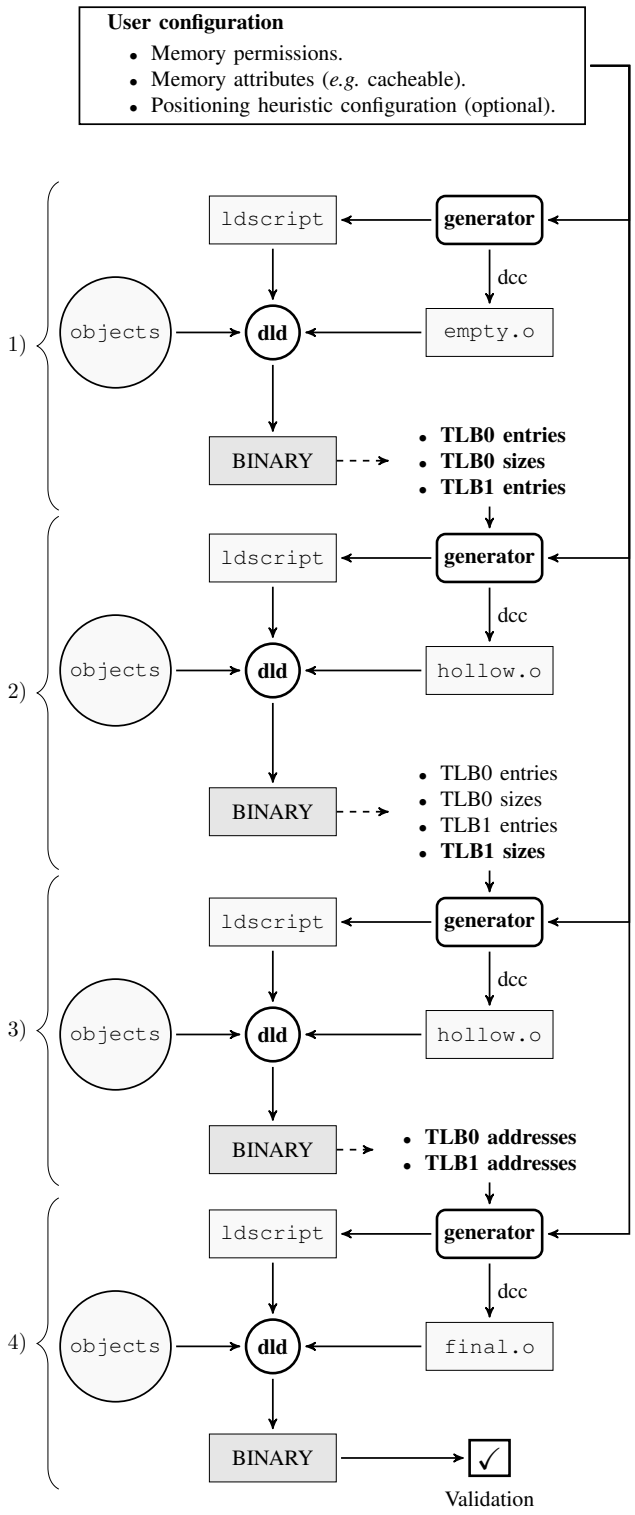

Fig. 3. Four-links process using the Wind River Diab compiler, implementing the generic method illustrated in Fig. 2. It enables the generation of an executable containing self-describing memory protection tables. Each link and compile operations are performed by off-the-shelf compilers. The objects on the left side are generated from the user application code, and are never modified.

and static memory protection tables describing how a fixed set of tasks are to be protected during the execution of a safety-critical system. Because the executable binary to be certified embeds self-describing tables, these can be verified off-line with appropriate tooling, reducing the certification efforts for equivalent run-time algorithms. As this method can be fully automated, we believe it contributes to reduce certification costs without compromising safety properties.

ACKNOWLEDGMENT

We would like to thank engineers from Krono-Safe who indirectly helped contributing to this paper by their discussions and implementations; François Guerret, Matthieu Texier, Amira Methni and Jean-Marc Lacroix for their reviews.

REFERENCES

[1] J. Windsor and K. Hjortnaes, "Time and space partitioning in spacecraft avionics," in *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*. IEEE, 2009, pp. 13–20.

[2] D. Haworth, T. Jordan, A. Mattausch, and A. Much, "Freedom from interference for autosar-based ecus: a partitioned autosar stack," *Automotive-Safety & Security 2012*, 2012.

[3] L. Rierson, *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2013.

[4] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[5] R. Achermann, "On memory addressing," Ph.D. dissertation, ETH Zurich, 2020.

[6] J. Brygier and M. Oezer, "Safety and security for the internet of things," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016.

[7] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Temporal isolation of hard real-time applications on many-core processors," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–11.

[8] J. S. Camier, C. Aussagues, M. Lemerre, and V. David, "A Toolchain For Designing And Implementing Efficient, Flexible And Safety-Critical Applications," in *Embedded Real Time Software and Systems (ERTS2008)*, Toulouse, France, Jan. 2008.

[9] S. Louise, M. Lemerre, C. Aussagues, and V. David, "The oasis kernel: A framework for high dependability real-time systems," in *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*. IEEE, 2011, pp. 95–103.

[10] V. David and J. Delcoigne, "Security method making deterministic real time execution of multitask applications of control and command type with error confinement," Nov. 20 2007, uS Patent 7,299,383.

[11] R. Thomas, "Lief - library to instrument executable formats," https://lief.quarkslab.com/, April 2017.

[12] R. RTCA DO-178 and EUROCAE, "Software Considerations in Airborne Systems and Equipment Certification," RTCA DO-178C, 2011.

[13] A. Methni, E. Ohayon, and F. Thurieau, "ASTERIOS Checker : A Verification Tool for Certifying Airborne Software," in *10th European Congress on Embedded Real Time Systems (ERTS 2020)*, Toulouse, France, Jan. 2020.

[14] *PowerPC e500 Core Family Reference Manual, Rev1*, NXP, september 2011, with errata.