TASA: Toolchain-Agnostic Static Software Randomisation for Critical Real-Time Systems

Leonidas Kosmidis[*,†], Roberto Vargas[*,†], David Morales[†],
Eduardo Quiñones[†], Jaume Abella[†], Francisco J. Cazorla[†,‡]

[*]Universitat Politècnica de Catalunya, Spain [†]Barcelona Supercomputing Center, Spain
[‡]Spanish National Research Council (IIIA-CSIC), Spain

ABSTRACT

Measurement-Based Probabilistic Timing Analysis (MBPTA) derives WCET estimates for tasks running on processors comprising high-performance features such as caches. MBPTA's correct application requires the system to exhibit certain timing properties, which can be achieved by injecting randomisation in the timing behaviour of the task under analysis. However, existing software-randomisation techniques require costly modifications in the industrial production toolchain (compiler, linker, runtime or hardware) in terms of development and certification. In this paper we present TASA, a new software randomisation tool that relies on source-code transformations of the application (i) requiring no changes in existing toolchains, which heavily reduces tool qualification and implementation costs; and (ii) achieving competitive WCET estimates that we assess on a gcc- and a llvm-based compilation toolchain on a real board.

1. INTRODUCTION

The system-level value added by software increases in every Critical Real-Time Embedded Systems (CRTES) generation across all CRTES sectors– e.g. avionics [8], railways [9] and automotive [7]. For example, in the automotive domain, the number and complexity of functions, e.g. X-by-wire, control for combustion engines, obstacle detection and collision avoidance, has steadily increased during the last years [4]. Efficiently running this software results in a relentless growth in processing needs [7].

Providing the required processing power rests on the use of high-performance hardware features, such as caches, which however 'disrupts' industrial practice for timing analysis [30]. In particular, measurement based timing analysis– acknowledged as the most commonly used technique in different CRTES domains [30] – finds difficulties in providing tight and reliable WCET estimates [2]. Measurement-based *Probabilistic* Timing Analysis (MBPTA) [17] is a new timing analysis that helps in analysing complex hardware. MBPTA deploys *Extreme Value Theory* [16] (EVT), a well established mathematical tool to model the extreme of distributions that for CRTES is used to obtain a distribution of the worst-case execution time of programs, known as probabilistic WCET (pWCET).

MBPTA has shown promising results in providing pWCET estimates for software running on complex hardware such as unified and multiple levels of caches [21] and has already been positively assessed with automotive [19] and avionics [11, 12] case studies. MBPTA relies on the timing behaviour of the system under analysis to have a probabilistic nature. This can be achieved when the underlying hardware/software platform fulfils certain probabilistic properties, referred to as *MBPTA compliance* [22]. This requires those events generating variations in the execution time to be forced to work on their worst latency or to be time-randomised [22].

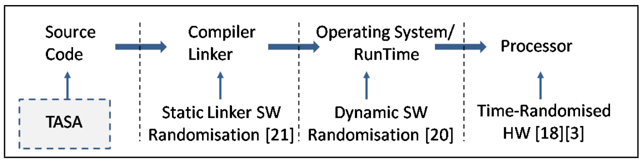

Source Code
Compiler Linker
Operating System/ RunTime
Processor
TASA
Static Linker SW Randomisation [21]
Dynamic SW Randomisation [20]
Time-Randomised HW [18][3]

Figure 1: Randomisation approaches and the level of the production toolchain pipeline that they impact.

The latter is achieved via specific hardware, for instance, randomised cache placement and replacement [20]. However, customised hardware limits the adoption of this technology. At software level, several user-level libraries have been developed to randomly allocate program code and data across runs such that an argument can be made on the hit/miss probability of cache accesses, achieving MBPTA compliance. Unfortunately, current software randomisation techniques require changing system tool-chains, i.e. introducing modifications on the compiler and run-time libraries. This not only challenges – and heavily increases the cost of – tool qualification against safety standards such as ISO26262 [14] in the automotive domain, but it requires randomisation to be applied to each compiler and target. Figure 1 summarises the existing MBPTA enabling technologies, as well as the components they affect in the production pipeline.

In this paper we propose a certification-friendly *Toolchain-Agnostic Static Software randomisation Approach* (TASA), that randomises the location in which memory objects are defined within the source-code of the program. TASA relies on the direct relation between the order of memory objects in the source-code and their memory position, since compilers typically generate the elements of the executable (code, data, etc.) in the same order they are encountered in the source file. Therefore, by adding functionally-neutral padding code and data and reorganising the declaration of variables and functions across runs, TASA reaches the randomisation properties as previous approaches with the following advantages: (1) TASA does not require system toolchain to be modified, as randomisation is applied at source code level. TASA tool qualification is not repeated for each compiler and target – as previous randomisation solutions require – but instead once per programming language (standard). This significantly reduces TASA qualification costs and hence facilitates the adoption of software randomisation. (2) its application at source code level makes TASA portable across platforms and compilers. The only compiler-dependent element of TASA is activating certain flags to prevent some compiler optimisations, which are usually deactivated in real-time systems and have a minimum impact on performance.

We evaluate TASA on a real processor featuring high-performance components and on top of two compilers to demonstrate its transparency to different system toolchains. Our results with the EEMBC Automotive benchmarks show that TASA maintains compliance

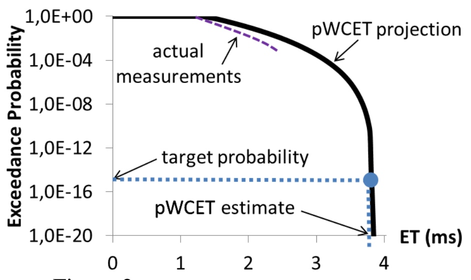


Figure 2: Example application of MBPTA.

with MBPTA – passing all required statistical tests –, with small impact on memory footprint due to the functionally-neutral padding and the reorganisation variables declaration (7% increase), preserves average performance and provides low pWCET estimates, outperforming existing software randomisation techniques.

2. BACKGROUND AND RELATED WORK

While traditional Deterministic Timing Analysis methods derive a single WCET estimate per task [30], MBPTA [17] produces a distribution of WCET estimates (pWCET) that describes the probability of one instance of the task to exceed a given execution time. This provides the system designer flexibility to select an appropriate pWCET value according to the system requirements reducing over-provisioning in the timing estimates. For instance, an ASIL-D component in the ISO26262 standard that executes 1,000 times an hour, requires pWCET estimates below 10^{-11} so that the probability of a timing failure for the task is below 10^{-8} per hour [1].

MBPTA works on a set of end-to-end execution time measurements (hundreds or few thousands of runs per program [17]) collected from the execution of a program on the target hardware. MBPTA processes them with EVT [16], see Figure 2. EVT is a statistical method able to predict the minimum or the maximum of a probability distribution. EVT handles execution time observations that can be modelled with *independent* and *identically distributed* (i.i.d.) random variables, which can be assessed with the corresponding statistical tests [16] (more details in the results Section). These properties can be provided via the adoption of time-randomised hardware and software means [10, 22, 3, 18, 19]. Interestingly, the use of software randomisation techniques [18, 19] provides the means to enable the use of MBPTA on conventional hardware increasing its applicability. It is important to remark that MBPTA cannot be applied directly on conventional deterministic hardware, where its requirements are not met due to the deterministic nature of the events that take place (e.g. whether an access hits/misses in cache).

Problem Statement. Program's memory layout affects the performance of modern processors [23][26][28]. This occurs because both, linking order of object files [28] and small changes of the memory layout (e.g. environmental variables), may significantly affect programs' execution time. Building on the latter observation, [6] used software randomisation to isolate this effect from the actual performance benefit of various compiler optimisations.

For CRTES several software randomisation approaches have been proposed. *Dynamic Software Randomisation (DSR)* [18][6] performs the randomisation at runtime during the initialisation phase of the program, so the location of objects in memory is randomised across different executions of the program. To that end, DSR combines a compiler pass that modifies appropriately the intermediate representation of the application's code and a run-time system, based on self-modifying code, that is in charge of performing the relocation of objects in memory. The memory objects whose location is randomised are code, stack frames and global data[2]. Previous work [19] showed that DSR generated code cannot be used in the automotive domain for both practical and certification reasons. First DSR cannot be used with the automotive microcontrollers in which read-only code and data live in flash memories. Moreover, DSR makes intensive use of pointers and dynamic objects which complicates the certification of DSR-randomised software across the standard ISO26262 [14] which requires (among others) a limited use of pointers, no use of dynamic objects, and no hidden data or control flow.

Link-level Static Software Randomisation (LL-SSR) [19] achieves the same effect as DSR in an entirely static manner. Since the position of memory objects in the executable defines their placement in main memory [28], and therefore the cache layout, LL-SSR carries out all relocation operations statically at compile and link time. LL-SSR generates a number of different binaries of the same program each with different random allocation of memory objects in the executable. By randomly selecting an executable from the pool of the generated ones, the timing properties required for the pWCET estimates are preserved. Similarly to DSR, LL-SSR randomises code (functions), stack frames and global data.

Interestingly LL-SSR is certification aware, regarding the software being compiled but not with respect to the qualification of tool which implements it (compiler and linker). In the automotive sector for example, the ISO26262 (Chapter 8) puts in place new tool qualification means. Based on whether a tool error may produce a violation of a safety requirement and how probable is to predict/identify such errors, the tool is assigned a Tool Confidence level (TCL). The tool's TCL and the ASIL of the software produced by the tool determines the appropriate tool qualification method. Other CRTES domains have similar tool qualification needs, which needs to be addressed for high criticality software.

As will be demonstrated in next section, in terms of qualification, the benefits of TASA over LL-SSR are a follows: (1) TASA tool takes as input a source code and delivers a randomised platform-independent source code, while LL-SSR tool takes a source code as input and delivers a binary platform-dependent code. Hence TASA has to be qualified on a programming language (standard) rather than per compiler and platform as it is the case of LL-SSR. It is understood that integrating TASA tool into a specific safety lifecycle and performing integration tests with the underlying compiler is significantly simpler than testing the updated compiler versions implementing LL-SSR changes. (2) LL-SSR has to be implemented for each compiler and target (backend), since the changes on the stack are backend dependent, while TASA is implemented once per programming language (standard). This heavily reduces implementation costs. (3) TASA' compilation toolchain dependence reduces to activating some flags to prevent some compiler instrumentations. This is arguably simpler and lighter (in terms of qualification and implementation costs) than the changes required by LL-SSR in the compilation tool chain. And (4) unlike LL-SSR, which required modifying compiler internals, TASA can be applied to proprietary compilers in which the source code is not available. Further, although there exist well-established open-source compilers, i.e. gcc and llvm, modifying them to support LL-SSR is complex and it is hard (and costly) to find skilled developers.

3. TASA

[1]ASIL-D components is commonly compared to SIL-3 defined in the IEC61508 standard [13] (upon which the ISO26262 is built) that defines the probability of failure per hour of components.

[2]DSR allows random allocation of heap objects. However, the use of dynamic memory allocation is not allowed in CRTES [25][1] and therefore it is not considered in this paper.

```
int a(int i)
{
  int m=5;
  return i+m;
}

void b(int j)
{
  char k[2]={1,0};
  double f= a(j)+k[0];
}
```

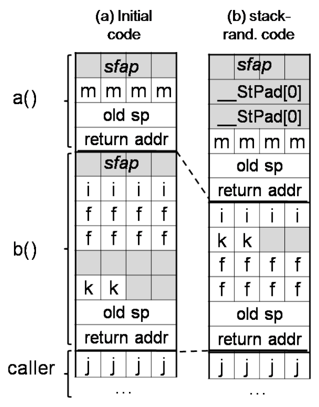
(a) Initial Code Fragment

```
int a(int i);
void b(int j);

void b(int j){
  char k[2]={1,0};
  double f= a(j)+k[0];
  return;
  asm volatile("nop");
  asm volatile("nop");
  return;
}
int a(int i){
  int m=5;
  return i+m;
  asm volatile("nop");
  return 0;
}
```

(b) Code-randomised code

```
int a(int i)
{
  volatile double
      __StPad[1];
  int m=5;
  return i+m;
}

void b(int j)
{
  double f;
  char k[2]={1,0};
  f= a(j)+k[0];
}
```

(c) Stack Rand. Code



(d) Stack frame layout for (a) and (c). Alignment padding shown in gray.

Figure 3: Code fragments showing code and stack randomisation scenarios.

Software randomisation adds a random padding among memory objects and reorders them from run to run resulting in a randomised MBPTA-analysable execution time. *The challenge lies on doing so while keeping certification friendliness, requiring no change in the target-domain tool chain, keeping MBPTA compliance assessed by passing i.i.d tests, and keeping the competitive edge in terms of tight pWCET estimates and reduced memory overheads.*

TASA applies *source-code level static software randomisation* (SL-SSR) to enable MBPTA on conventional caches. Similarly to LL-SSR, the integration of TASA and MBPTA requires TASA generating several images – each with a random allocation of memory objects – that are run on the target platform. Execution times are collected and used to feed MBPTA that delivers a pWCET estimate. At operation, one of those randomly generated images is used [19]. TASA requires some compiler optimisations to be disabled, incurring minimum impact on average performance as discussed in this section and experimentally evaluated in Section 4.

3.1 Executable Structure

This section presents the internals of a binary file and how it is related to the source code. Although TASA's principles apply to any binary format, we consider the *elf binary* format [29] for the sake of the discussion in this paper. An elf executable comprises four sections: `.text`, `.rodata`, `.data` and `.bss`. Before the program is executed, the linker loads each section in memory, at its corresponding address specified in the executable. The `.text` section contains the executable code of the program. The `.rodata` section contains all read-only program data, which includes global and local static (i.e. local variables that retain their values across function calls) variables declared as `const`, and string literals defined anywhere in the program. Both segments are placed consecutively so that in systems which implement memory protection mechanisms, the entire range of these addresses is write-protected. Moreover, in CRTES sectors where flash memories are used, such as in the automotive case, these segments are assigned to those read-only memory devices. The `.data` section has the initialised global and local static variables. Finally, the `.bss` section contains global and local static variables which are not initialised or their initial value is zero.

Besides these sections, the binary loader creates at runtime two additional special segments: `heap` and `stack`. The former provides space for dynamically allocated objects and it starts after the end of `.bss` section. The latter provides space for each function's stack frame, where its local variables and arguments live, and it starts at the highest address of the memory space and grows towards the `heap` (lower addresses). The `heap` segment is ignored in this study since in CRTES dynamic allocation is not allowed.

The elements that compose the executable's sections cannot be arbitrarily placed in memory but certain architectures require the address of memory accesses to be always aligned to the size of the access (e.g. MIPS, SPARC), while others exhibit significant performance penalty when this alignment is not followed (e.g. x86, PowerPC). Hence, compilers generate code that complies with the alignment requirements of the target platform (unless instructed otherwise). For instance, the starting address of each stack frame is aligned to the maximum access size, called *stack frame alignment* (usually 8 bytes), and its size is rounded up to be multiple of this size. This stack frame size adjustment can create empty space called compiler-introduced *stack frame alignment padding (sfap)*. Likewise, each variable is allocated in a memory position multiple of the alignment size as well, e.g. a double precision variable (8 bytes) must be allocated in a memory address multiple of 8, while a `char` variable has no placement requirements since every address location is multiple of 1 byte. This empty space between consecutive variables is called *alignment padding (ap)*. This padding plays a fundamental role in our proposal.

3.2 Code Placement Randomisation

TASA performs code randomisation via two mechanisms.

Function ordering. Code randomisation relies on the fact that, by default, the location of functions in the binary takes place in the same order that they are encountered in the source file, unless certain compiler optimisations are enabled [23]. As a result, randomising the order of functions in the source file achieves the goal of randomising the placement of the functions in the object file. This is illustrated in Figure 3. In the original file (Figure 3a), function `a()` precedes function `b()`, so the compiler generates code in the same order in the object file. Figure 3b, shows the corresponding randomised code, in which the functions are swapped. Moreover, in order to guarantee correct cross function dependencies, function prototypes are introduced at the beginning of the file (if not present).

Function Padding. Although random function reordering provides different mappings in the instruction cache, the number of different cache layouts is limited by the number of function permutations. This is mitigated by artificially increasing the size of each function by a random *padding*. TASA adds padding in the form of an arbitrary number of additional instructions at end of the function as shown in Figure 3b. The number of added instructions ranges from 0 up to the number of instructions that fit in a cache way since this covers all the potential mappings of instructions to cache sets. The new inserted instructions have no functional or timing effect on

```
const char msg[]="OK";
long l;
long m=3L;
unsigned int f;
const double pi=3.14;

int c(void){
  static int i=1;
  if(i==MAX){
    i=0;
    printf("reset");
  }
  return i++;
}

void d(int i){
  static long time;
  static long ticks=1L;
  time = clock();
  ticks++;
  printf("%l␣ms", time);
}
```

(a) Initial Code Fragment

```
const double pi=3.14;
const char msg[]="OK";
long m=3L;
unsigned int f;
long l;

void d(int i){
  static long time;
  static long ticks=1L;
  time = clock();
  ticks++;
  printf("%l␣ms", time);
}

int c(void){
  static int i=1;
  if(i==MAX){
    i=0;
    printf("reset");
  }
  return i++;
}
```

(b) Global & Static Variable rand.

Figure 4: Code fragments under various data rand. scenarios.

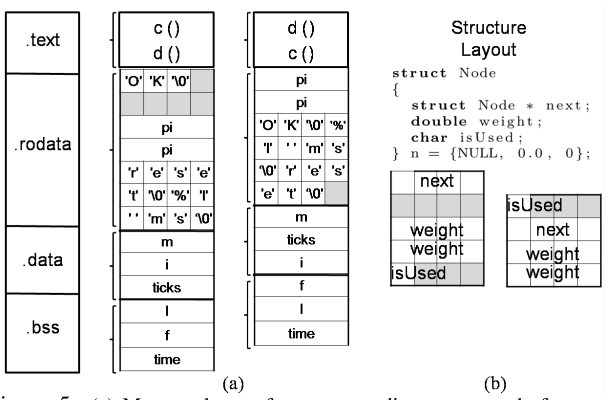
the executed code, which is achieved by using *nop* instructions. C programs can directly call assembly instructions using the `asm` directive. While in general using assembly instructions limits source code portability, note that this particular instruction does not harm the portability of the technique due to its ubiquitous nature. Moreover, the `volatile` qualifier in the asm instruction is required to prevent the compiler from removing the introduced assembly instructions during its optimisation passes. Further, in order to avoid that the introduced instructions affect functions' execution time, we ensure that the new inserted code is never reached by introducing a `return` statement (if not present in the code) with a return value compatible with function's return type (e.g. function `b()`). In the case that the function features more than a single return point, this padding is applied only to the last one, since the purpose is to pad the size of the function, without any functional effect. This solution requires dead code elimination and unreachable code elimination optimisations of the compiler to be disabled, otherwise the added code would be automatically removed by the compiler.

3.3 Stack Frame Randomisation

We achieve stack randomisation through 2 different complementary and combinable methods.

Stack Padding. Since the size of the stack frame is determined by the number and the size of the function's local variables and the arguments of the functions that it calls, we can randomise the stack by introducing a randomly sized local array in the list of local variables, see function `a()` in Figure 3c. We use the *volatile* qualifier to instruct the compiler not to perform any optimisation on these variables. In order to effectively increase the stack frame, even when the frame size is rounded up to the alignment size, the array is declared as `double`. Similarly to code placement randomisation, the size of the array, and therefore the padding introduced, is randomly sized up to the size of a cache way.

Figure 3d shows the stack frame of `a()` after applying stack randomisation. In the original code, the stack frame contained only a 32-bit variable plus the always included return address and the old stack pointer. Assuming a 32-bit environment, the size each of the return address and the stack pointer is 4 bytes, therefore the compiler rounds up the stack frame size to 16 bytes, in order to comply with the stack frame alignment requirements, instead of using a 12 byte stack frame. In the randomised scenario, the stack frame also holds a double variable increasing the stack frame size, which with stack alignment padding reaches 24 bytes size. Note that if instead of double, the type of `__StPad` was float or integer, the desired effect would not be achieved, and in both cases the stack frame of `a()` would have the same size.



```
struct Node
{
  struct Node * next;
  double weight;
  char isUsed;
} n = {NULL, 0.0, 0};
```

Figure 5: (a) Memory layout for corresponding source code fragments from Figure 4. (b) Struct memory layout.

Local Variable Declaration Order. The stack can also be randomised by changing the declaration order of local variables, as they are allocated on the stack based on their position from the stack pointer. Such a fine-grain stack randomisation, which cannot be achieved by the existing software randomisation solutions, is of particular interest because it achieves randomisation of *intra-object* [18] conflicts (*intra-stack* in particular). Figure 3d shows the impact that the order in which local variables are declared in function `b()` defined in Figure 3c has on the stack frame. The compiler introduces an alignment padding (sfap) between `k` and `f` to ensure that `f` is placed on an address multiple of 8. Such alignment padding, however, may not be required when shuffling the local variable declaration as shown in Figure 3d.

3.4 Program Data Randomisation

Global data, including variables defined in the global scope, static variables defined in each function and string literals, reside in the corresponding sections `.rodata`, `.data` and `.bss`, depending on whether they are constants and have/lack initial values.

Figure 5a shows the content of sections `.text`, `.rodata`, `.data` and `.bss` for the initial code fragment shown in Figure 4a, including the corresponding alignment. Despite compiler's optimisations, the location of each variable in the binary is heavily dependent on its *relative position in the source code file with respect to the variables mapped in the same binary section.* In other words, swapping the order of two variables will only affect the mapping if it changes their relative position regarding the rest of the variables mapped in the same section. For example swapping variables `l` and `m` in the source code shown in Figure 4a would not change the mapping in the binary which is displayed in Figure 5a as they reside in different data sections, `.bss` and `.data` respectively. On the other hand swapping `msg` and `pi` would, as both variables reside in the same section (`.rodata`).

TASA guarantees that variables from different sections are shuffled by grouping variables belonging to the same section first before shuffling each group individually. Similarly to the stack frame, changes in the order of symbols in the sections may produce changes in the alignment padding. This affects the position (and size) of the subsequent variables in the same section and the position of the symbols in the following sections. Figure 4b shows the source code after applying program data randomisation, while its corresponding memory layout is shown on the right part of Figure 5a.

Static variables and string literals: The position of these types of variables in the corresponding section is bound to the position in the source code of the function they are declared in, e.g. as shown in the string literals used in printf in Figures 4 and 5. If `c()` and `d()` were not swapped, their position in `.rodata` would remain unchanged. The same effect happens for static variables `i` and `ticks`. Hence, static variables and strings cannot be randomised freely, unless code randomisation is enabled too. This is an important difference with LL-SSR [19], where these variables can be shuffled independently from the code of their function. However, in applications with many functions and static data, the memory layout is not meaningfully affected by this binding.

Compound Structure Randomisation. Existing software randomisation solutions do not manage efficiently compound constructs, like programming language structures (Figure 5b). This is because the access structures' member variables are allocated in a relative position (offset) from the starting address of the structure, based on their declaration order in the structure definition. This offset is hard-coded into the binary, and so neither DSR nor LL-SSR can modify structures' member positions. Instead, TASA can shuffle structures' members in order to randomise data access patterns when used in arrays, and intra-object conflicts, provided the size of the structure is bigger than a cache line. Similarly to stack randomisation (see Section 3.3), due to members' alignment padding, the size of each structure depends on the particular ordering of its members, which in turn affects the memory layout. This is illustrated in the example of Figure 5b. This process requires the aggregate initialisator (if any) to be shuffled respectively and the `__packed` attribute, which prevents alignment padding, to be absent from the the structure's definition.

In general, any hard real-time program that follows the corresponding programming guidelines [25][1] for these systems, is amenable to this randomisation. However, this is not true in general for code that makes assumptions about the particular placement of fields in memory, and accesses the structures without using the names of the fields. This type of non-portable code is usually found in low level software such as device drivers and the use of structure randomisation must be disabled as it would lead to wrong code. In this case, this type of randomisation needs to be disabled.

3.5 Multi-source binaries

When a program with multiple sources is compiled, each file is first compiled to produce an object file. Each object file is comprised by the same sections as the final executable. During the linking phase, these sections are joined together, in the order that the object files are passed to the linker, to form the sections of the final executable.

Hence, if TASA were applied in each source file independently, the elements of each section could only be randomised with respect to the elements in the source file in which they are declared. Moreover, in the case that two source files used a common definition of a structure and the compound structure randomisation were enabled, the code accessing the structs in each file would use a different structure definition. To avoid the above problems, TASA merges all files in a single source file, taking into account *symbol linkage*. Then the previously described randomisations are applied with global scope.

3.6 Compiler Optimisations

In order TASA to be applied, a few compiler optimisations need to be disabled as identified in the previous sections. Concretely, function/data reordering and dead-code (unreachable) code elimination.[3] These optimisations can be divided into two groups: a) performance optimisations and b) size optimisations. All but one optimisation, fall in the latter group. In particular, the only performance enhancing optimisation which is disabled is function reordering for performance. While this can be effective for small code, its effectiveness is reduced when the number and the size of functions is increased, which is the case in modern CRTES. Moreover, this process is NP-complete for a large number of functions [24], therefore can lead to extremely long compilation times. For this reason, this optimisation is typically disabled in such systems. The rest of the optimisations have no performance cost, but memory size cost. These optimisations try to find function and data ordering that minimises the overall memory footprint. Similarly, these optimisations do not scale well for large code bases, increasing significantly the compilation times.

Disabling dead-code/data and unreachable code elimination, if they are effective, impacts size only. Safety critical systems must execute only code that is verified at deployment, therefore dead code is eliminated from the application before TASA is applied. Moreover, the code and data padding introduced by TASA are non-functional elements that are never executed or used. Thus, this constraint is still met. Section 4.1 shows that disabling those optimisations on EEMBCs has no performance or memory impact.

[3]Note that in code/stack randomisation `volatile` only prevents the removal of the TASA introduced padding.

4. EVALUATION

In this section we show that TASA provides competitive WCET estimates w.r.t. DSR. Further, the latter generates code that uses pointers and dynamic code making difficult its certification, while it cannot be used in domains where read-only flash memories are employed such as the automotive [19]. Regarding LL-SSR, TASA is functionally equivalent and provides exactly the same results as LL-SSR (for this reason we do not provide comparison results). Overall, TASA provides a better implementation than LL-SSR tool without incurring LL-SSR's high implementation and qualification costs described in Section 2.

We implement TASA as a source-to-source compiler for ANSI C implementing all randomisation techniques described in Section 3. To assess the independence of TASA w.r.t. the industrial toolchain considered, we use two different compilers to compile the TASA-transformed code: gcc-4.4.2 and vanilla llvm 3.1. We use the EEMBC Autobench benchmarks [27], which comprise features similar to real applications in the automotive domain.

Platform. Processors used in CRTES cover a wide set of ISA and processor models from Freescale, Renesas, Infineon, Texas Instruments, etc. In this paper, we focus on an FPGA version of a Sparc v8 LEON3 [5] processor developed by Cobham Gaisler, used in the hard real-time domain. It features an advanced memory hierarchy including 4-way set-associative instruction cache with 128 sets per way and 32-byte cache lines, and a 4-way set-associative data cache with 256 sets and 16 bytes per cache line. The DSR, upon which TASA compares, is available on this platform (including the support on Sparc compilers). In order to limit the memory overheads of TASA we select the maximum random padding to be equal to $1/20^{th}$ of each cache way size, that is at most 50 instructions and stack padding 50 doubles.

Methodology. For each benchmark we perform 1,000 passes of the original source code with TASA, in order to generate an equal number of binaries with different layouts. In [17] it is shown that 1,000 runs are enough for EEMBC to safely apply MBPTA. Each binary is executed once and its end-to-end execution time is collected.

We compare TASA with the state-of-the-art software randomisation tool for dynamic software randomisation (DSR), Stabilizer [6]. The compiler pass of the original open source tool works only on LLVM 3.1, for this reason we used that old version of LLVM. For each benchmark we perform all software randomisation techniques as in [18] and collect 1,000 end-to-end execution times.

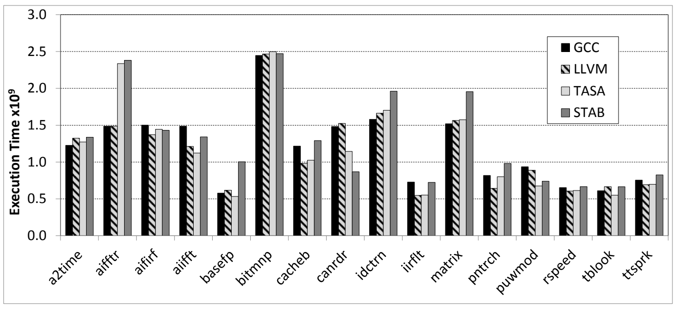


Figure 6: Average execution time measured in processor cycles for TASA and DSR (STAB).

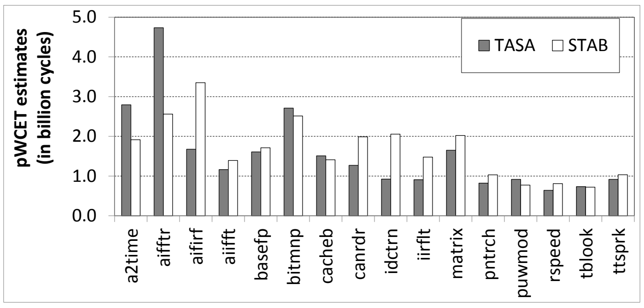


Figure 7: Worst Case Execution Time for TASA and Stabilizer

Dealing with jittery instructions. Previous works in the MBPTA literature [22][3] consider hardware modifications in instructions with low jitter such as the floating point instructions, in order to operate in their maximum latency during the analysis phase of the system. This way, they ensure that the execution conditions at the operation phase of the deployed system match or upperbound the ones observed during the analysis phase, which is a fundamental MBPTA requirement. In our platform, based on the free version of LEON3 which doesn't include a floating point unit (FPU), the software is compiled with software emulation of floating point operations. In this case, all instructions have the same latency at both analysis and operation, and therefore no special treatment is required. In case that an FPU was available, a simple calculation over the number of observed jittery instructions to account for the worst case latency that may be exhibited at operation would be sufficient to comply with this MBPTA requirement.

4.1 Average Performance

Besides WCET, average performance is important in CRTES to optimise non-functional metrics such as power and energy. Our results, shown in Figure 6, confirm that on average both TASA and Stabilizer provide average execution times close to the default generated code by GCC and LLVM, with TASA being on average within 0.4% of LLVM's average performance.

In two cases, the average performance of both randomised configurations is significantly different to that of the non-randomised one such as `aifftr` (worse) and `canrdr` (better). This is related to the default memory layout selected by the compiler, which happens to be very good or very bad compared to the set of potential memory layouts, which randomisation can explore. Finally in almost all cases, the execution times of Stabilizer are longer than for TASA, with significant differences at times (`basefp`, `matrix`). The reason is due to Stabilizer's execution time overhead: at start-up the runtime performs some required operations before the execution of the program such as the code relocations, while other additional stack-randomisation are performed at runtime. Additionally, the increase of the `.bss` section due to the introduced metadata (see Section 4.3) induces a significant performance degradation issue, since this section needs to be zeroed-out before the execution of the system. While this operation is optimised in desktop systems by techniques like copy-on-write, in an embedded system without operating system like our target, it cannot. Therefore the bigger the `.bss` is, the longer this operation takes.

Several compiler flags are used to maintain transformations implemented by TASA: (1) `-fno-toplevel-reorder` instructs the compiler to generate functions and globals in the order encountered in the binary; (2) `-fno-section-anchors` prevents placing static variables used in the same function, in nearby locations in order to be accessed with a single base; (3) `-fno-dce` disables dead-code elimination; (4) `-Wnounreachable-code` instructs the compiler to respect the introduced unreachable code padding; and (5) `-Wno- unused-variable` preserves the stack randomisation padding. We have observed that optimisation disabling with gcc has neither effect in memory consumption nor in performance. For llvm the result was exactly the same; none of the three dead code elimination (dce) passes (dce, advanced dce and global dce) removed any dead or unreachable code[4]. This can be explained because EEMBC have a small/controlled codebase which does not contain any dead code.

4.2 pWCET estimates and MBPTA compliance

For pWCET computation we use the MBPTA method described in [17]. Instead of showing the pWCET curve as presented in Figure 2, which is infeasible for space constraints, we show the pWCET estimates obtained with the different techniques for a cut-off probability of 10^{-15} per run. This value has been chosen since it has been shown appropriate for applications with the highest integrity level, i.e. SIL-4, defined in the IEC61508 standard [13], and upon which the ISO26262 is defined.

The safe application of MBPTA requires the obtained execution time observations to be modellable with i.i.d. variables [17]. This can be assessed by using the corresponding statistical tests. In particular we use the Box-Ljung test [15], which is more robust than the Runs test used in [17] for independence and the Kolmogorov-Smirnov test for identical distribution. For a significance level of $\alpha = 0.05$ the results of both tests have to be above 0.05 not to reject the i.i.d. hypothesis. For all EEMBC benchmarks the results of the i.i.d. tests for TASA are above 0.05, confirming that the execution time observations taken from the different binaries, each with a randomised layout, can be modelled with i.i.d variables, making TASA compliant with MBPTA.

Figure 7 shows the pWCET for EEMBC benchmarks, at a cut-off probability of 10^{-15}, with all possible randomisations enabled w.r.t. the actual execution time on LLVM with no randomisation. For most benchmarks TASA provides lower pWCET than Stabilizer (6% on average across all benchmarks), following the same trend as for average performance, due to the start-up and runtime overhead of dynamic software randomisation. However, in few cases (`a2time`, `aifftr`) the pWCET of TASA is higher than Stabilizer's one despite the lower average performance. This occurs because TASA can explore some rare memory layouts (3% of the explored ones) with much higher execution time than the average one, that Stabilizer was not able to generate, due to its coarser-grain randomisation.

4.3 Memory Overheads

Figure 8 shows the memory consumption increase for the `.text`, `.data` and `.bss` segments for EEMBC Automotive when using

[4]To our knowledge LLVM does not reorder code/data.

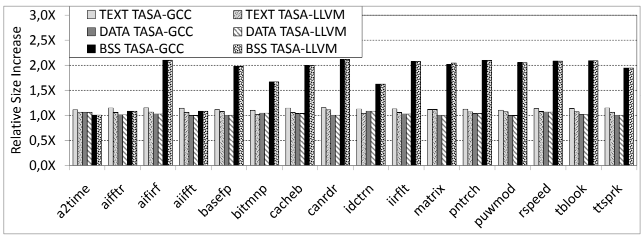


Figure 8: Memory overhead for different binary sections. Results normalised to gcc or llvm without software randomisation. Values for TASA are average for all binaries.

TASA with gcc and llvm, which we call *TASA-GCC* and *TASA-LLVM* respectively. Each software-randomised setup is normalised to the same non-randomised configuration, e.g. TEXT TASA-GCC provides the relative increase in the text segment when TASA is applied over the non-randomised one, when in both cases gcc was used for compilation.

Code Segment. Code padding increases application's code footprint. TASA overhead is less than 1.13% on average when used with gcc (*TEXT TASA-GCC*), applying a random padding up to 50 nop instructions per function (200 bytes). Using TASA with LLVM (*TEXT TASA-LLVM*) this overhead is only 7%, because the generated code with gcc is more compact, therefore the relative increase is smaller.

Data Segments. For the *stack* size, results not shown in Figure 8, TASA increases the space used per function around 25% on average, using a stack padding size up to 50 doubles (400 bytes). When local variables randomisation is enabled there is a variability of the stack size of $\pm 1\%$. For the `.data` section, the overhead of TASA is less than 7% for both toolchains (Figure 8). For the `.bss` TASA' overhead is 80% on average, but its contribution to the total footprint is small, since it represents less than 10% of the memory footprint in the our benchmarks. The reason for the increase is that EEMBCs contain large uninitialised arrays which are used to simulate data output. Note that in a real system those arrays would not exist. Randomising the location of these large arrays results in big size differences due to the alignment padding, as we have explained in Section 3.

Putting it all together. Overall, TASA incurs a 7% increase on the memory footprint of EEMBC Automotive benchmarks for both compilers.

5. CONCLUSIONS

MBPTA's promising results can only be obtained on those platforms adhering to MBPTA's requirements. Prior attempts to make systems compatible with MPBTA require changes in different parts of the toolchain stack, such as the compiler, linker, runtime or even hardware, making more difficult its adoption in industrial environments due to additional costs related to the development and certification of the modified toolchain. We propose TASA, a software randomisation technique that works at source code level, which makes it portable across platforms. Our results using EEMBC Autobench benchmarks running on a real COTS processor from the hard real-time domain, show that TASA provides similar results to DSR, which generates code difficult to certify due to its use of pointers and dynamic constructs. Further, while TASA and LLSSR pWCET results are the same, the former considerably reduces the qualification and implementation costs due to its tool-chain independence.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROXIMA Project (grant agreement 611085), the Ministry of Science and Technology of Spain under contract TIN2015-65316-P and the HiPEAC Network of Excellence. Leonidas Kosmidis is funded by the Spanish Ministry of Education under the FPU grant AP2010-4208. Jaume Abella is partially supported by the Spanish Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

6. REFERENCES

[1] JPL Institutional Coding Standard for the C Programming Language. *Jet Propulsion Laboratory, CalTech*, 2009.

[2] J. Abella, C. Hernández, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015.

[3] C. Hernández, J. Abella, F. J. Cazorla, J. Andersson, A. Gianarro. Towards Making a LEON3 Multicore Compatible with Probabilistic Timing Analysis. In *Data Systems In Aerospace (DASIA)*, 2015.

[4] R. Charette. This car runs on code. In *IEEE Spectrum Online*, February 2009.

[5] Cobham Gaisler. *Leon3 Multiprocessing CPU Core.*

[6] C. Curtsinger and E. D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[7] G. Darren Buttle, ETAS GmbH. Real-Time in the Prime-Time. *ECRTS-12 Keynote Talk.*

[8] G. Edelin. Embedded systems at Thales: the ARTEMIS challenges for an industrial group. *Lecture at ARTIST Summer School*, 2009.

[9] F. Corbier, L. Kislin, E. Forgeau. How Train Transportation Design Challenges can be addressed with Simulation-based Virtual Prototyping for Distributed Systems. In *3rd European Congress - Emdedded Real-Time Software (ERTS)*, 2006.

[10] F. J. Cazorla et al. PROARTIS: Probabilistically Analyzable Real-Time Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):94:1–94:26, May 2013.

[11] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013.

[12] F. Wartel et al. Timing analysis of an avionics case study on complex hardware/software platforms. In *Design, Automation & Test in Europe (DATE)*, 2015.

[13] International Electrotecnical Commission (IEC). *IEC 61508-3. Functional safety of electrical / electronic / programmable electronic safety-related systems - Part 3: Software requirements*, 2010.

[14] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[15] J. Abella, J. Castillo, F. J. Cazorla, M. Padilla. Extreme value theory in computer sciences: The case of embedded safety-critical systems. In *6th International Conference on Risk Analysis (ICRA)*, 2015.

[16] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.

[17] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[18] L. Kosmidis, J. Abella, E. Quiñones, F. J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe (DATE)*, 2013.

[19] L. Kosmidis, J. Abella, E. Quiñones, F. J. Cazorla. Multi-level unified caches for probabilistically time analysable real-time systems. In *34th IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[20] L. Kosmidis, C. Curtsinger, E. Quiñones, J. Abella, E. Berger, F. J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *Design, Automation and Test in Europe (DATE)*, 2013.

[21] L. Kosmidis, E. Quiñones, J. Abella, G. Farrall, F. Wartel, F. J. Cazorla. Containing timing-related certification cost in automotive systems deploying complex hardware. *Proceedings of the 51st Annual Design Automation Conference (DAC) , Best Paper Award*, 2014.

[22] L. Kosmidis, Quiñones, J. Abella, T. Vardanega, I. Broster, F. J. Cazorla. Measurement-based probabilistic timing analysis and its impact on processor architecture. *17th Euromicro Conference on Digital System Design (DSD)*, 2014.

[23] S. McFarling. Program optimization for instruction caches. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1989.

[24] E. Mezzetti and T. Vardanega. Towards a cache-aware development of high integrity real-time systems. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2010.

[25] MISRA. *Guidelines for the Use of the C Language in Critical Systems*. 2013.

[26] N. Gloy, T. Blackwell, M. Smith, B. Calder. Procedure placement using temporal ordering information. In *30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1997.

[27] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[28] T. Mytkowicz, A. Diwan, M. Hauswirth, P. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[29] Tool Interface Standard (TIS). *Executable and Linking Format (ELF) Specification*.

[30] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, May 2008.