Containing Timing-Related Certification Cost in Automotive Systems Deploying Complex Hardware

Leonidas Kosmidis[*,†], Eduardo Quiñones[†], Jaume Abella[†],
Glenn Farrall[⊙], Franck Wartel[Ψ], Francisco J. Cazorla[†,‡]
[*]Universitat Politècnica de Catalunya, Spain [†]Barcelona Supercomputing Center, Spain
[⊙]Infineon Technologies, UK [Ψ]AIRBUS, France [‡]Spanish National Research Council (IIIA-CSIC), Spain

ABSTRACT

Measurement-Based Probabilistic Timing Analysis (MBPTA) techniques simplify deriving tight and trustworthy WCET estimates for industrial-size programs running on complex processors. MBPTA poses some requirements on the timing behaviour of the hardware/software platform: execution times of end-to-end runs have to be independent and identically distributed (i.i.d.). Hardware and software solutions have been deployed to accomplish MBPTA requirements. The latter has achieved the i.i.d. properties running on some commercial off-the-shelf (COTS) processor designs. Unfortunately, software randomisation challenges functional verification needed for certification since it introduces indirections through pointers in the code. In this paper we propose a new approach to software randomisation able to contain its functional verification costs. Our approach performs software randomisation *statically*, as opposed to current dynamic approaches. We carefully review the requirements of the new approach and prove its feasibility.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems; C.4 [**Performance of Systems**]: Performance attributes

General Terms

Design, Performance, Standardization

Keywords

Real-time, WCET, Certification

1. INTRODUCTION

In the automotive market, the number and complexity of functions (e.g. steer-by-wire, control for combustion engines, automatic emergency-braking triggered by collision avoidance systems, etc.) has steadily increased during the last years [6]. Increasingly complex hardware (i.e. cache hierarchies, multicores, etc.) is used to deliver the performance required by those complex functions. The use of more complex hardware (and software), however, challenges the application of conventional timing analysis techniques. Static timing analysis (STA) (1) has been shown to have scalability issues for industrial-size programs [16]; and (2) STA is highly sensitive to the lack of information about hardware and software details (e.g., the addresses of data accesses) resulting in a rapid increase of worst-case execution time (WCET) estimates. Meanwhile, measurement-based timing analysis (MBTA) relies having the ability to take detailed measurements of the system and being able to test the system adequately. There are many forms of MBTA including end-to-end testing and RapiTime analysis [20]. For end-to-end testing where the code is simply measured end to end, there is often a lack of confidence in the testing to exercise the worst case path, so a safety margin may be added on top of the longest execution time observed. However, the soundness of such approach is hard – if at all possible – to prove, especially on top of processors deploying caches in which abrupt execution time variations can appear [17].

Probabilistic timing analysis (PTA) [4, 9, 5, 7] addresses some of the issues of STA and MBTA. PTA yields *probabilistic WCET* or pWCET estimates, which stand for time bounds with an associated exceedance probability (e.g., 10^{-15} per run). The measurement-based variant of PTA, MBPTA [7], which is the focus of our paper, has been proven to successfully obtain trustworthy and tight pWCET estimates even for industrial-size applications [21].

MBPTA poses some requirements on the hardware/software platform ensuring that the execution times observed from the execution of a program in the platform have some probabilistic properties. In particular, the execution times of end-to-end runs of the program under analysis must be modellable with independent and identically distributed (i.i.d.) random variables. This has been proven feasible, either by deploying new hardware designs that introduce randomisation in the timing behaviour of some components, like the cache [12]; or by delivering software randomisation [13], which can provide the required properties on top of some commercial off-the-shelf (COTS) processor designs. Software randomisation relies on placing memory objects (code and data) in random memory locations so that their placement in cache becomes also random despite the deterministic nature of conventional cache memories, which typically implement modulo placement and least recently used (LRU) replacement. This is achieved by randomly placing memory objects at program start time and/or when they are created dynamically, and accessing them through tables to deal with indirections. Unfortunately, the use of indirection tables in *dynamic randomisation* has been shown to challenge functional verification of programs needed for certification against safety standards like ISO26262 for automotive [11].

In this paper we propose a new approach to software randomisation in which randomisation is performed *statically*. While dynamic software randomisation (DSR) relies on including some randomisation code in the program executable (binary) so that, every time the program is invoked, memory objects are randomly placed, static software randomisation (SSR) relies on generating several binaries for the same program. In each binary memory objects are shifted appropriately to produce the same effect as if those objects are placed at random, yet statically determined, locations at runtime. By using SSR, modifications on the binary introduce neither indirections nor extra pointers. As a consequence, functional verification remains no

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2593112.

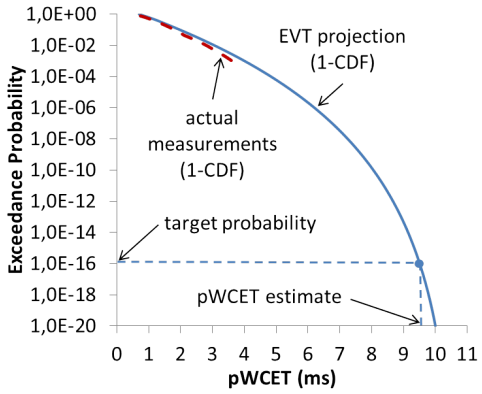

Figure 1: Example of $1-CDF$ and MBPTA projection.

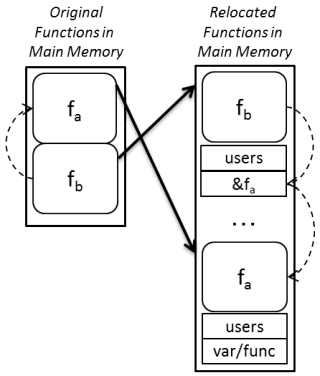

Figure 2: Relocation table is placed adjacent to the end of functions f_a and f_b into the main memory.

more complex than without software randomisation, requiring a qualified compiler that generates the SSR binaries. Analogously to DSR, SSR allows computing the probability that a given arrangement of objects in memory leads to a timing violation.

2. BACKGROUND

2.1 Probabilistic Timing Analysis (PTA)

PTA [4, 9, 5, 7] provides WCET estimates with an associated probability of occurrence, called probabilistic WCET (pWCET) estimates, which can be exceeded with a given probability, thus leading to a *timing failure*. Similar to the behaviour of hardware, which may fail with a given probability, PTA extends the notion of probability of failure to timing correctness. PTA can obtain pWCET estimates for arbitrarily low probabilities. For instance, the pWCET value associated to an exceedance probability of 10^{-20} per hour of operation may be chosen. This is large below the probability of random hardware failures, which must be proven to be below 10^{-8} per hour of operation for the highest integrity level in ISO26262 (ASIL-D).

In this paper we focus on MBPTA as it is close to industrial practice to compute WCET estimates, where measurement-based approaches are common practice. MBPTA derives pWCET estimates for a program based on a collection of end-to-end observed execution times on a time-randomised architecture for which an execution time profile (ETP) can be derived for each instruction. MBPTA applies Extreme Value Theory (EVT) [14], a well-known statistical method, that based on the complementary cumulative distribution function (1-CDF) provides the probability that the execution time of a given program run exceeds a given pWCET estimate. The underlying principle of MBPTA is shown in Figure 1 in which the dotted line is tail distribution of the observed execution times and the continuous line represents the upper-bound provided by MBPTA.

In order to apply MBPTA it is required that the execution times collected from a program have a probabilistic behaviour such that it can be modelled with random variables, in particular with i.i.d. random variables[1]. Under PTA the probabilistic timing behaviour of an instruction can be represented with an Execution Time Profile (ETP). An ETP describes the discrete probability distribution function of execution times, that is, the different execution times of the instruction and their corresponding probabilities. Having one ETP for each dynamic instruction allows the required i.i.d properties to emerge at the level of end-to-end measurement runs [2].

[1]Two random variables are: (a) independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other [10]; (b) identically distributed if they have the same probability distribution [10].

2.2 Dynamic Software Randomisation (DSR)

DSR relies on placing the code of functions and their stack frames, as well as global and static variables in random memory locations on each invocation.

Code randomisation (DSR-code): The randomisation of functions is performed by relocating each function's code into a random memory location at program startup time. A *Relocation Table* (RT) is placed at the end of each new relocated function to identify the addresses of all globals and functions accessed by the relocated function (see Figure 2). Stabilizer's [8] compiler transformation rewrites all accesses to globals and other functions to indirect accesses through the RT. Figure 2 shows an example of two functions f_a and f_b with the latter calling the former (dotted line). The Figure also shows the layout of the functions once reallocated. Function relocation is carried out in two phases:

- *Initialisation-Relocation.* At program startup, the Stabilizer runtime library allocates, for each function and in a random location, a sufficiently large block of memory from the DieHard [3] allocator and copies its code to this location. The runtime then generates a RT adjacent to the new function location, with all entries pointing to the original locations of any called functions and globals.
- *Redirection.* Then, Stabilizer overwrites the first instruction of the original function location with a static jump to the new location. This forwards all future calls to the random function location.

Stack Randomisation (DSR-stack): Stabilizer randomises the stack by making it non-contiguous. Stabilizer allocates a block of memory *for every <function,nesting level> pair*, with the allocation being done at function call time. Hence, if under the main function we have the sequence of calls $f_a(); f_b(); f_a();$, on the first call to $f_a()$ Stabilizer allocates a block, which is not deallocated and reused for the second call to $f_a()$. If a function is called nestedly (e.g., $f_a()$ calls $f_a()$), Stabilizer allocates a new block of memory for every recursion level.

Every function has a depth counter and frame table that maps the depth to the corresponding stack frame (see Figure 3). The depth counter is incremented at the start of the function and decremented just before returning. On every call, the function loads its stack frame address from the frame table. If the frame address is null, Stabilizer runtime allocates a new frame.

Global/static Variables Randomisation (DSR-globals): Although not described in [13], globals[2] location can be randomised analogously to function code at program startup and access to globals is then performed through indirections and pointers.

[2]We refer to global and static variables as *globals* for the sake of convenience.

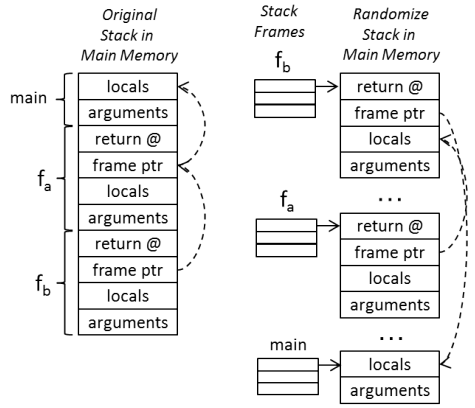

Figure 3: Randomisation of stack frames of functions f_a and f_b into the main memory.

3. STATIC SOFTWARE RANDOMISATION

This section introduces how DSR challenges functional verification, and how this problem is addressed by our SSR techniques. We also review the implications of performing SSR.

3.1 Functional Verification of Software

Programs implementing safety-related functions need to meet ISO26262 standard [11] requirements in the automotive domain. Rigor is needed to meet standard requirements, and thus, a number of software design principles are listed in those standards to facilitate a successful software functional verification. Among those principles, in the ISO26262 we can find the following ones: (i) a limited use of pointers, (ii) recommendations against the use of dynamic objects, and (iii) no hidden data flow or control flow. This is particularly true for the highest safety levels (e.g. ASIL-C and ASIL-D).

DSR challenges functional verification of software in different ways. First, indirections for functions, stack frames and global/static variables occur through pointers. Second, functions (code), globals and stack frames use dynamic objects allocated by the DieHard [3] allocator. And third, data and control flow occurs through pointers, thus making it non-obvious.

As a consequence, generating use cases for software testing is challenging as the analysis of boundary values and error guessing, as listed in ISO26262, is hard when pointers and dynamic objects are used. Furthermore, the fact that functions are copied dynamically in new memory locations, thus writing code in data memory segments for its later execution, makes programs have self-modifying code, which some architectures may not support as the memory management unit may prevent memory pages in the data segments from being fetched for their execution.

3.2 Static Code Placement Randomisation (SSR-code)

Conventional *DSR-code* relies on placing functions (code) at random locations every time the program is run. For that purpose, functions are compacted in the binary (no empty space between functions) and they are copied to the desired random locations when the program is run. Such an approach increases the memory space needed in the data segments at runtime to copy the code, but the size of the binary is increased negligibly to include the function copy code and indirect function calls.

In the case of our new *SSR-code*, random locations for functions are determined statically at compile time and such locations are already reflected in the function layout in the binary. This could be done by simply introducing some random shift (padding) between functions; however, this would increase binary size inordinately. In order to achieve the same effect efficiently, a number of steps must be taken to place functions randomly while minimising the size of the binary. Those steps are described in the algorithm in Figure 4.

(1) **Input:** F function list
(2) WS = cache way size (bytes)
(3) LS = cache line size (bytes)
(4) $F_{place} = \emptyset$
(5) **for** $i = 1$ **until** $cardinality(F)$ **do**
(6) $\quad pad = \left(random()\ mod\ \frac{WS}{LS}\right) \cdot LS$
(7) $\quad F_{place} = F_{place} \bigcup \{F, sizeof(F), pad\}$
(8) **end for**
(9) $Align = 0$
(10) $Binary$ = empty file
(11) **While** $F_{place} \neq \emptyset$ **do**
(12) $\quad MinPad = WS$
(13) $\quad$ **for** $i = 1$ **until** $cardinality(F_{place})$ **do**
(14) $\quad\quad Waste = \left(F_{place}(i) \rightarrow pad + WS - Align\right)\ mod\ WS$
(15) $\quad\quad$ **if** $Waste \leq MinPad$ **then**
(16) $\quad\quad\quad Best = F_{place}(i)$
(17) $\quad\quad\quad MinPad = Waste$
(18) $\quad\quad$ **end if**
(19) $\quad$ **end for**
(20) $\quad Binary = append(Binary, MinPad, Best)$
(21) $\quad Align = (Best \rightarrow pad + Best \rightarrow size)\ mod\ WS$
(22) $\quad F_{place} = F_{place} - Best$
(23) **end while**
(24) **Return:** $Binary$

Figure 4: Algorithm to randomly place functions in the binary.

We assume a cache deploying modulo placement as it is one of the most common placement functions. Given a cache of CS bytes with W ways, the size of a way is given by $WS = \frac{CS}{W}$. Given a function whose initial (instruction) address is $@_A$, its first instruction will be placed in cache set $S_A = \left\lceil \frac{@_A \bmod WS}{LS} \right\rceil$ where LS is the line size in bytes. Note that, if we shift the initial address by WS, $@_A + WS$, the first instruction will still be placed in S_A. Further, no assumption is made regarding the cache line alignment of instructions. Their original alignment with respect to the cache line is preserved when introducing a random shift if its size is a multiple of the cache line size.

Given the function list (F), we traverse it computing a random offset (or padding) for each function that determines its alignment w.r.t. the beginning of the cache way (lines 5-8). This padding (*pad* in the figure) is a multiple of the cache line size[3]. F_{place} is a tuple containing the function, its size and the padding assigned to that function.

Once each function has its own padding (offset), any given function k_i can be described as a contiguous memory area whose first instruction is mapped to set $S_{k_i}^{init}$ and whose last instruction is mapped to set $S_{k_i}^{last}$. The algorithm then looks for the function k_0 whose $S_{k_0}^{init}$ is the closest to 0 (or even 0). Once this function is placed in the binary, the algorithm looks for a function, from the list of not-yet-allocated functions, k_1, whose initial set is as close as possible to the set in which the last instruction of k_0 was assigned (lines 13-19). That is, the algorithm reduces the wasted space between $S_{k_i}^{last}$ and $S_{k_{i+1}}^{init}$ (lines 15-18). Note that the problem of minimising the total padding in-between functions is a NP-complete problem. In the algorithm, once each function has its own offset with respect to the first set of a cache way, we initialise the next alignment available (line 9) that can be used to place a function and the binary (line 10). Then, we traverse the set of functions, which includes their respective sizes and paddings (line 11). We initialise the minimum padding found to the largest value possible in line 12 (the size of a cache way). For each function in the

[3]We assume that functions are cache line aligned as this improves spatial locality, although *SSR-code* is not limited to this assumption.

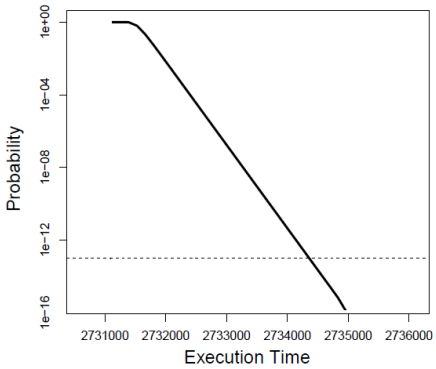

Figure 5: pWCET distribution for an industrial program expressed in processor cycles.

set (line 13) we compute how much space would be wasted in between this function and the last one placed in the binary if we placed the current function next (line 14). If such wasted space is equal or lower than the smallest one found (line 15), then we pick the current function as the best candidate to be placed next (line 16) and update the minimum padding found (line 17). Once all functions have been examined and the best candidate found, we append such function to the binary adding the corresponding padding (line 20), update the alignment desired for the next function to be placed (line 21) and remove the function placed from the set (line 22). This process repeats until all functions are placed and the layout of all functions in the binary is obtained. The algorithm we propose for *SSR-code* is a greedy algorithm whose cost is $O(N^2)$ w.r.t. the number of functions.

Once the functions are conveniently arranged in the binary, their initial addresses are fully known and can be used for calling them, as in a non-randomised program. Therefore, indirections needed for *DSR-code* are no longer needed. Moreover, since functions do not have to be written into the data memory space, as it is the case for *DSR-code*, which relies on self-modifying code, memory protection is not challenged. Instead, instructions are fetched only from the code segment.

3.3 Static Stack Frame Randomisation (SSR-stack)

While functions exist in the binary, stack frames are created dynamically. In a non-randomised binary, the stack frame for a new function call is created right after the last stack frame. *DSR-stack*, instead, allocates stack frames in random locations and uses them to break such determinism. However, the location of those stack frames is unknown statically.

To make the location of randomly allocated stack frames known at compile time, we propose *SSR-stack* which, just before allocating a new stack frame, adds a random padding in the stack frame, computed at compile time. The added padding, as for the case of functions, ranges between 0 and the size of a cache way (WS) since address placement in cache wraps up beyond the cache way size. By doing so, stack frame placement is random and different across functions with *SSR-stack* as it is the case for *DSR-stack*. Note that, unlike *DSR-stack*, *SSR-stack* imposes that nested calls to the same function use the same (random) padding. Although this decreases the degree of randomisation (nested calls use the same padding value), padding is still random across different images (binaries) so i.i.d. properties across images still hold as needed by MBPTA.

SSR-stack adds to each function an instruction that decreases the stack pointer by padding bytes. Since the value of padding is known at compile time, no extra pointers are used and all addresses can be determined as in a non-randomised binary, thus not increasing functional verification complexity.

3.4 Static Global/Static Variable Randomisation (SSR-globals)

As indicated before, the case of globals is analogous to that of functions. Therefore, the algorithm in Figure 4 can also be used to randomise the location of globals. If globals are placed in different segments (i.e. `.data` and `.bss` segments), then the algorithm must be applied individually to each segment. Still, the complexity of the problem is the same as for code since objects characteristics are fully known at compile time and hence, their location in the binary can be randomised analogously.

4. DEPLOYING DSR AND SSR

In this section we study the impact of deploying DSR and SSR in an automotive computing system, which requires understanding the pWCET estimates obtained with MBPTA. Figure 5 shows the pWCET obtained for a representative program [21]. The X-axis shows time, with the scale not starting at 0, and the Y-axis probabilities in logarithmic scale. The function represents the pWCET estimate for the program. We observe that the pWCET function has a steep gradient. This means that the increase in pWCET experienced when decreasing the cutoff exceedance probability is small. For instance, the increase in pWCET from 10^{-12} to 10^{-16} is less than 1%[4].

4.1 DSR

With DSR, code and data placement are randomised across executions of the binary of a program, so that the exceedance probabilities of the pWCET estimates obtained for that program apply at the end-to-end run granularity. For instance, an exceedance probability of 10^{-16} implies that an execution of the program can exceed the corresponding pWCET estimate with at most that probability for that execution instance alone. In order not to exceed a timing failure rate per hour (e.g., 10^{-16}), if the program is executed, for instance 10^3 times per hour, the system designer should choose as pWCET estimate the value at exceedance threshold 10^{-19}, so that it is guaranteed probabilistically that the accumulated timing failure rate of all instances of execution of the program in one hour is below 10^{-16}.

4.2 SSR

With SSR, code and data placement are randomised across images (binaries). Thus, i.i.d. properties are attained at *image level* rather than at *end-to-end* run granularity. Thus, timing failures apply at the granularity of binary instead of at end-to-end run granularity. In order to derive pWCET estimates with MBPTA, during the analysis phase, SSR deploys an automated approach in which N experiments are run, each with a different binary. The collected execution time from each run is fed to MBPTA to derive a pWCET estimate. Note that N is in the order of hundreds as shown in previous studies [7, 21].

For the system realisation (as opposed to analysis) there are two different approaches possible for SSR: (i) deploying a different binary in each system or (ii) deploying one of such binaries in *all* systems.

SSR with different binaries per system unit: In this approach the probability of timing failure of the program per system unit deployed is independent across units. If the pWCET of the program is not to be exceeded with a probability higher than 10^{-22} and 10^6 units are delivered, there is a probability of 10^{-16} of exactly one system in which that program binary experiences timing failures and 10^{-32} that it would happen in two different units. The downside of this approach is that it

[4]To appreciate how small the 10^{-16} cutoff probability is, consider that Extinction Level Events (ELE), such as an asteroid hitting the Earth, are estimated to happen about once every 100 million years, hence at an arrival rate of 10^{-16} per second, or 10^{-12} per hour.

implies each unit having a different binary[5], which may not be acceptable if each individual unit is not fully tested.

SSR with the same binary: In this approach a single binary is generated with SSR and deployed in all systems. Then, if the binary exceeds the pWCET, it may do so *in all units*. However, this can be made to occur with negligible probability, in the same order of probability of an Extinction Level Event (ELE) to occur. Hence, if for instance, the pWCET is not to be exceeded with a probability higher than 10^{-16} it is much more likely to experience an ELE than a timing failure.

5. EVALUATION

In this section we evaluate the overheads introduced by SSR. In order to place objects into random memory locations we consider *Stabilizer*, an existing compiler transformation and runtime system that enables random placement of functions, stack frames, heap objects, and globals for C and C++ programs [8].

The Stabilizer compiler pass has been developed within the LLVM compiler [1]. Each source code file is first compiled to LLVM's internal representation. The resulting byte-code files are then processed with LLVM's optimisation tool running the Stabilizer compiler pass. The optimised byte-code is then compiled to the target machine's object code (PowerPC in our case). Since SSR is performed statically, there is no need to link any runtime library as it is the case for DSR. Finally, performance overheads have been obtained on a cycle-accurate execution-driven simulator based on the SoCLib simulation framework [18] modelling a 4-stage pipelined processor with a memory hierarchy composed of first level separated instruction and data caches (1 cycle hit, 100 cycles miss), and main memory. Both instruction and data caches are 8KB 8-way set-associative with 32B lines. Both caches implement modulo placement and LRU replacement policies. The data cache implements a write-through, no-allocate write policy.

5.1 Memory Overheads

5.1.1 SSR-code *and* SSR-globals

SSR-code and *SSR-globals* are the main source of memory overheads when applying SSR. As shown in Section 3.2, some space is left between functions (and also between globals) in the binary so that their placement in cache is random. Next we analyse the overheads incurred when applying *SSR-code* to an industrial-size (case-study) application [21][6]. This application consists of around 5,000 functions whose sizes range between few bytes and 300KB. The total size of those functions is 4.7MB if they are enforced to be aligned with cache line boundaries assuming a cache line size of 32B. To analyse the sensitivity of *SSR-code* to cache way size we consider cache way sizes of 1KB, 2KB, 4KB and 8KB. Table 1 reports the average size in bytes of the code segment obtained for each way size for 1,000 different static software randomisations of the application together with the relative size increase w.r.t. the original code segment size. Maximum values are relatively close to the average , thus proving the stability of our approach. Also, as the way size increases, inefficiency also increases because the average size of the padding between different functions in the binary also grows.

The same analysis can be applied to globals as for functions in the industrial application. In this case the application consists of around 70,000 globals whose sizes range between few bytes and 24KB. Those globals are all in the `.bss` segment, as it is initialised to zero values. Uninitialised segments such as the `.data` one are not recommended for safety-critical applications.

Table 1: Binary size overheads for the case study.

Way size	1KB	2KB	4KB	8KB
Code	4,866,386	4,923,387	5,040,105	5,288,843
% increase	2.8%	4.0%	6.5%	11.7%
Data	673,145	693,159	734,608	815,765
% increase	4.1%	7.2%	13.6%	26.1%
Code+data	5,539,531	5,616,546	5,774,713	6,104,608
% increase	2.9%	4.4%	7.3%	13.4%

The total size of those globals is 2.3MB if they are enforced to be aligned with cache line boundaries assuming a cache line size of 32B and only 640KB otherwise. In order to reduce such inefficiency we have packed globals so that they fill full cache lines and therefore, they can be kept cache line aligned. This leads to 18,000 objects, but most of them fill just one cache line. Small objects are prone to leave many gaps in between in the binary that cannot be filled. In particular, for a 1KB way size we observe a 60% data segment increase with 32B objects. Thus, we have packed globals into larger objects (1KB each) whenever possible for a total of 590 objects by simply packing the largest objects that do not exceed the cache line size. This still leads to around $32^{590} \approx 10^{900}$ different potential object placements for a 1KB way size[7], so the degree of randomness attained is still huge. Note that the degree of randomness has been proven not to affect the correctness of the approach but the value of the pWCET estimates if few different object placements can be obtained, which is not our case at all [13]. Note that by packing globals into sizes not exceeding the cache way size, no conflict can occur across those globals and such behaviour holds both at analysis time and at deployment. Results are shown in Table 1. Overheads are similar to those for *SSR-code*.

Total results for the binary, including code and data segments are also reported in Table 1. As shown, binaries are expected to grow little due to SSR, with cache way sizes in this range.

Sensitivity Analysis: We have also performed a sensitivity study considering random function sizes between 128B and 2048B varying the number of functions between 10 and 1,000. Results are shown in Figure 6. As for the case study, inefficiency grows for larger cache ways. Furthermore, we also observe that the larger the number of functions, the lower the relative overhead since it is easier to find functions to be placed with little padding. In particular if cache way size is 1KB the binary size overhead decreases from 19.2% (10 functions only) down to 2.0% (1,000 functions). Conversely, large cache ways are particularly harmful when few functions are placed since large padding cannot be avoided in-between those functions. For instance, for an 8KB way size the binary size is in the range of 2.5 to 3 times the original one if only 10 functions are placed. However, as the number of functions increases such overhead decreases. Finally, note that this sensitivity analysis for functions (Figure 6) is also valid for globals as they are analogous problems. Thus, the larger the number of globals or the larger their size, the lower the relative overhead due to SSR.

5.1.2 SSR-stack

Regarding *SSR-stack*, it is not stored in the binary. Instead, when functions are called some space may be wasted in the stack due to stack frame padding. As for functions and globals, the largest padding must be smaller than the cache way size. Thus, the overhead is incurred dynamically when functions are called and it can be upper-bounded by the maximum function call depth[8] in the program multiplied by the cache way size. Since programs implementing safety-critical functions

[5]As binaries are randomly generated, it can be the case that different units get identical binaries, although this is highly unlikely to occur.

[6]We have corroborated that the characteristics of this avionics application reasonably resemble some real automotive applications in terms of function count and function size.

[7]The number of atoms in the Universe is estimated to be 10^{80}.

[8]The maximum stack depth refers only to the user code stack, excluding system stack which is not modified by our method.

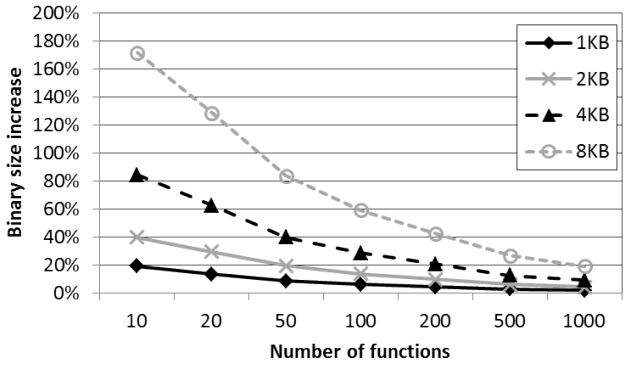

Figure 6: Binary size overheads for the sensitivity study varying function number between 10 and 1,000 and cache way size between 1KB and 8KB.

do not use complex call structures [21], call depth is typically low (largely below 10 nested calls), thus wasting little memory space. For instance, for a maximum call depth of 5 functions and a cache way size of 2KB, up to 10KB of memory would be wasted dynamically (5KB on average) for *SSR-stack*.

If such overhead is regarded as excessive due to RAM memory availability constraints, the random padding could be added just once at the beginning of the program. The effect would be analogous to packing globals: determinism introduced (all stack frames consecutively placed) remains identical at analysis and deployment, but the degree of randomisation is reduced, thus potentially increasing pWCET estimates.

Note that by randomising at least the starting address of the stack, any further call, including interrupts, will get its stack in a random location. It will not be random w.r.t. other stack frames, but this does not challenge PTA as the states observed at analysis time match those at deployment.

5.2 Performance

DSR has been shown to have moderate or low execution time overheads [13]. SSR introduces fewer instructions (1 per stack frame) than DSR (several for each function/stack indirection) and does not require any initialisation process, as opposed to DSR. Hence, SSR reduces DSR overheads. In particular, SSR execution time overheads are largely below 1% w.r.t. non-randomisation even for programs with a large number of function calls to small functions – the worst scenario for SSR due to stack frame padding.

6. RELATED WORK

The advent of Probabilistic Timing Analysis (PTA) [9, 4, 5, 7] facilitates the computation of WCET estimates on top of complex architectures, but poses some requirements on the underlying hardware/software platform. Those requirements can be met by hardware means [12] or by software means on top of COTS hardware [13]. While hardware solutions are more efficient, no processor has been deployed yet providing those features, so software solutions based on software randomisation are the only way to apply PTA on today's systems.

Software randomisation has been used in the context of security [15] and bug tolerance [3], as well as in the context of WCET estimation together with PTA [13]. In all cases such software randomisation has been performed dynamically.

While timing guarantees obtained with PTA have been proven to fit functional safety standards [19], DSR has some side effects on the functional verification of software since extra pointers and indirections take values unknown at analysis time. To the best of our knowledge, our proposal on static software randomisation is the first solution reconciling PTA on COTS hardware and affordable functional verification.

7. CONCLUSIONS

PTA delivers trustworthy and tight WCET estimates, but poses some requirements on the underlying platform that can be achieved on top of COTS hardware if software randomisation is delivered. Unfortunately, software randomisation challenges functional verification of software against safety standards due to the use of pointers and indirections.

In this paper we propose a simple solution to override those limitations of *dynamic* software randomisation by providing *static* software randomisation. We show how static software randomisation can be implemented, identifying its advantages (affordable functional verification, simple implementation) and its limitations (some extra storage space required).

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 (www.proxima-project.eu), grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence and the EU COST Action IC1202: Timing Analysis On Code-Level (TACLe). Leonidas Kosmidis is funded by the Spanish Ministry of Education under the FPU grant AP2010-4208.

8. REFERENCES

[1] LLVM. `http://dragonegg.llvm.org/`.
[2] J. Abella et al. Measurement-Based Probabilistic Timing Analysis and i.i.d property. White Paper. 2013. `http://www.proartis-project.eu/publications/MBPTA-white-paper`.
[3] E.D. Berger and B.G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.
[4] G. Bernat, A. Colin, and S.M. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS*, 2002.
[5] F.J. Cazorla et al. PROARTIS: Probabilistically analysable real-time systems. *ACM TECS*, 2013.
[6] R.N. Charette. This car runs on code. In *IEEE Spectrum*, 2009.
[7] L. Cucu-Grosjean et al. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *ECRTS*, 2012.
[8] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. In *ASPLOS*, 2013.
[9] J. Hansen et al. Statistical-based WCET estimation and validation. In *WCET Workshop*, 2009.
[10] W. Feller. *An introduction to Probability Theory and Its Applications*. John Willer and Sons, 1996.
[11] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[12] L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.
[13] L. Kosmidis et al. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.
[14] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.
[15] L. Lixin, J.E. Just, and R. Sekar. Address-space randomization for Windows Systems. In *ACSAC*, 2006.
[16] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. In *WCET Workshop*, 2011.
[17] E. Mezzetti et al. Cache Optimisations for LEON Analyses. Technical Report COLA-FR-001-i1r1, ESA/ESTEC, 2011.
[18] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.
[19] Z. Stephenson et al. Supporting industrial use of probabilistic timing analysis with explicit argumentation. In *INDIN*, 2013.
[20] Rapita Systems. RapiTime Explained, White Paper MC-WP-001-17. `http://www.rapitasystems.com/system/files/RapiTime_Explained_White_Paper_web_ready.pdf`.
[21] F. Wartel et al. Measurement-Based Probabilistic Timing Analysis: Lessons from an Integrated-Modular Avionics Case Study. In *SIES*, 2013.