HAL
open science

Attack detection through monitoring of timing deviations in embedded real-time systems

Nicolas Bellec, Simon Rokicki, Isabelle Puaut

To cite this version:

Nicolas Bellec, Simon Rokicki, Isabelle Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems. ECRTS 2020 - 32nd Euromicro Conference on Real-Time Systems, Jul 2020, Modena, Italy. pp.1-22, 10.4230/LIPIcs.ECRTS.2020.8 . hal-02559549

HAL Id: hal-02559549

https://inria.hal.science/hal-02559549v1

Submitted on 30 Apr 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Attack detection through monitoring of timing deviations in embedded real-time systems

Nicolas Bellec
Univ Rennes, Inria, CNRS, IRISA, France
nicolas.bellec@irisa.fr

Simon Rokicki
Univ Rennes, Inria, CNRS, IRISA, France
simon.rokicki@irisa.fr

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA, France
isabelle.puaut@irisa.fr

Abstract

Real-time embedded systems (RTES) are required to interact more and more with their environment, thereby increasing their attack surface. Recent security breaches on car brakes and other critical components have already proven the feasibility of attacks on RTES. Such attacks may change the control-flow of the programs, which may lead to violations of the system's timing constraints.

In this paper, we present a technique to detect attacks in RTES based on timing information. Our technique, designed for single-core processors, is based on a monitor implemented in hardware to preserve the predictability of instrumented programs. The monitor uses timing information (Worst-Case Execution Time - WCET) of code regions to detect attacks. The proposed technique guarantees that attacks that delay the run-time of any region beyond its WCET are detected. Since the number of regions in programs impacts the memory resources consumed by the hardware monitor, our method includes a region selection algorithm that limits the amount of memory consumed by the monitor. An implementation of the hardware monitor and its simulation demonstrates the practicality of our approach. In particular, an experimental study evaluates the attack detection latency.

2012 ACM Subject Classification Computer systems organization → Embedded hardware; Security and privacy → Embedded systems security

Keywords and phrases Real-time systems, security, attack detection, control flow hijacking, WCET estimation, hardware monitoring

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.23

1 Introduction

Real-Time Embedded Systems (RTES) are becoming more and more present in our lives (IoT devices, embedded processors in cars, among others). Real-time constraints for these systems need to be validated through estimation of Worst-Case Execution Time (WCET) of their tasks [24] and schedulability analysis techniques.

Recent attacks on RTES [13] [14] [15] have shown attackers' increased attention to these systems. In particular, the predictability of RTES helps the attacker figure out the timing to strike, making it easier to mount attacks. Improving the security of RTES is a challenging task. Indeed, the deployed techniques (Address Space Randomization, Stack Smashing Protection, Position Independent Executables to cite only a few of them [18,19]) must protect the system against attacks, while maintaining the system's predictability, and with limited overhead. However, the need for predictability of RTES is also a strength to protect the system against attacks. Information on the system behavior (typically WCET) is available offline, and can thus be used to detect anomalous behaviors resulting from attacks.

© Nicolas Bellec and Simon Rokicki and Isabelle Puaut;
licensed under Creative Commons License CC-BY
Euromicro Conference on Real-Time Systems (ECRTS).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this work, we propose to use the inherent predictability of RTES to protect them against control-flow hijacking attacks for single-core architectures. The new defense mechanism exploits fine-grained partial WCET estimations of the application to detect attacks. A custom hardware monitor is in charge of observing the execution time of code regions and raising the alarm if the region's execution time is larger than its analyzed WCET, determined offline. The hardware monitor has a local memory containing the list of regions to monitor, together with their associated WCET. It then observes the execution at every clock cycle, by reading the current cycle counter and the program counter (PC) to detect execution times of code regions that are higher than the region's WCET. The hardware monitor proceeds by snooping the PC and cycle counter, and this does not require any modification of the application under monitoring. The proposed technique can thus be used on legacy code.

The attacks detected by our technique are all those that divert a region of code from its original control flow for a duration larger than its WCET, whatever the source of the attack. In particular, *software attacks* caused by buffer overflows are supported[1]. The proposed technique also detects *fault attacks*, that inject faults into the system via physical access to the device, using optical or electromagnetic perturbations [22]. On the other hand, attacks that bypass checks in the code are not detected because they do not cause the observed run time of a code region to exceed the region's WCET. Similarly, attacks that are fast enough are not detected. The guarantee provided by the proposed method is the detection of all attacks that increases the timing of a region beyond its WCET, whatever their source.

The selection of the regions to be monitored has an impact on the latency of attack detection: the smaller the WCET of the monitored regions, the faster the attack detection. On the other hand, monitoring more regions also increases the memory required for the hardware monitor. Since minimizing resource consumption is essential in any system, the proposed technique comes with a region selection algorithm that selects the best regions to monitor under resource constraints for the hardware monitor. The algorithm selects regions such that all code is covered (to detect attacks whatever their location) and provides a guarantee on the latency of attack detection. In our experiments, the algorithm reaches the optimal attack detection latency by selecting 47% of all the existing regions, on average.

Compared to related work on using timing information for detecting attacks (e.g. [27]), our technique does not impact the system predictability, since the system under monitoring is not modified. In addition, a technique is proposed to automate the selection of monitored regions. Compared to techniques using watchdogs to protect the control flow and memory accesses of programs [12] the technique we propose does not need any program instrumentation. A deeper comparison with related work is proposed in Section 7.

In summary the contributions of this paper are threefold:

- First, we present an algorithm, executed at compile-time, to select a set of regions whose timing will be monitored at run-time. Together, the regions cover the entire code, such that attacks can be detected whatever their location. The algorithm provides guarantees on the attack detection latency (largest WCET of the monitored regions), and operates under hardware constraints for the monitor, in particular limited memory for region storage. The algorithm is proven to provide the optimal attack detection latency in the absence of hardware constraints.

[1] One may consider that buffer overflows cannot occur in systems using static WCET analysis, because static WCET estimation tools model the whole system memory and thus detect possible buffer overflows. However, static WCET estimation tools also accept user-provided annotations such as loop bounds. Such incorrect annotations can lead to vulnerabilities remaining undetected by the estimation tool.

- Second, we contribute to a hardware monitoring co-processor, that uses WCET information on code regions to detect attacks. The practicality of the proposed hardware monitor is demonstrated using a simulation of the co-processor running along with a Leon 3 processor.
- Third, we provide an extensive evaluation of the proposed technique (compile-time selection of regions and hardware monitor) on a set of benchmarks. We evaluate the impact of memory constraints on the guaranteed attack detection latency and evaluate the actual detection latency and hardware cost of the monitor.

The rest of this paper is organized as follows. Section 2 gives an overview of the proposed time-based program monitoring technique. Section 3 details the algorithm we propose to select the regions to monitor, under resource constraints. Section 4 then gives implementation details. Experimental results are provided in Section 5. We discuss the scope and limitations of the method in Section 6. Related work is described in Section 7. Finally we conclude in Section 8.

2 Time-based detection of control flow hijacking attacks

The basic principle of our approach is to detect control-flow hijacking attacks by detecting if the run-time of code regions exceeds their statically estimated WCET. This approach handles attacks that divert the control-flow of the program to subvert a task (which could then be used to launch further attacks [3]). The approach is implemented using a dedicated hardware mechanism that measures the number of cycles spent in code regions and raises an alarm when the statically estimated WCET of the region is exceeded, meaning that an attack is under progress.

Selecting all regions in a given binary program not only consumes monitor memory but it is also not always beneficial. For example, selecting a very short region (in terms of WCET) does little for the protection if there exists a longer selected region, the maximum attack detection latency then being the duration of the longer region. Consequently, our approach comes with a compile-time selection of regions that are useful to reduce the attack detection latency. Region selection guarantees that all attacks modifying the control-flow for at least a known *Maximum Attack Window* (MAW) are detected, and operates under memory constraints for the hardware monitor. Thus the MAW is an upper bound of the attack detection latency. Region selection is performed on the program binary with no code modification. The set of regions selected at compile-time has the following properties:

- It completely covers the application binary, every reachable instruction is included in at least one region of the set. This way, attacks can be detected regardless of their location in the code. This ensures that there are no unprotected instructions.
- All regions are perfectly nested: two regions are either completely disjoint or one is fully included inside the other, such that monitored regions can be represented as a tree. This allows a simple yet efficient implementation of the monitor, using a stack representing regions entered and not exited yet.
- Regions only have one entry and one exit edge, as further detailed in Section 3.1.

For each selected region, the hardware monitor measures the execution time spent in the region, ignoring cycles spent in any inner selected regions, and compares it against the worst-case possible duration. The WCET of a region excluding inner selected regions is called *Maximal Inner Duration* (MID) of the region. The MID of each region is computed at compile-time. The *Maximum Attack Window* (MAW) is the maximal MID of the set of

monitored regions. An illustration of the MID of regions is depicted in Figure 1 for a very simple example of two regions R_1 and R_0, with R_1 nested in R_0. Plain lines at the bottom of the figure represent the MID of the two regions. Dotted lines at the top of the figure represent the possible execution times measured at run-time, an execution time higher than the MID reflecting the occurrence of an attack.

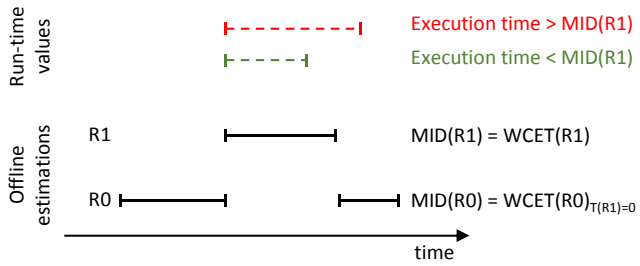


Figure 1 Maximal Inner Duration (MID) of nested regions.

Selecting the regions to monitor is a critical part of the proposed method, as it impacts the maximum attack window. The region selection algorithm we propose for solving this problem is presented in Section 3.

At run time, a dedicated hardware monitor measures the execution time of each region. As illustrated in Figure 2, the monitor has access to the program counter (PC) value of the processor at each cycle and uses it to determine whether the execution enters or exits a region. The monitor maintains a *stack* containing all regions currently active (entered and not yet exited) and the value of their cycle counters. The monitor only updates the cycle counter of the most nested region, currently under execution (stack top). The monitor uses two scratchpad memories: the *Stack Memory* (called SM in the following) is used to track active regions, and the *Region Memory* (called RM in the following) to store the tree of monitored regions.

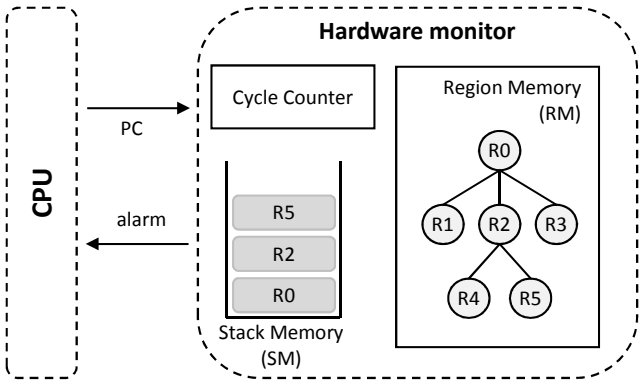


Figure 2 Overview of the monitoring system.

More precisely, at each cycle, the monitor successively tests if the following (non-exclusive) situations occur:

1. If the PC value corresponds to the exit point of the most nested region, this region is removed from the stack. The containing region then becomes the most nested region, and the cycle counter resumes from the value previously stored in the stack for that region.
2. If the PC value corresponds to the entry point of a new region, included in the most

nested region, the counter value of the most nested region is saved on the stack, and the monitor starts profiling the new region, with a counter starting at zero.

3. In all cases, the monitor increments the cycle counter of the most nested region and compares it with the value of the MID of the region. If it is greater than the MID, an attack is detected and an alarm is raised.

The monitor is implemented in such a way that it is able to analyze a new PC value at each cycle, and thus does not induce any slowdown in the execution of the application.

Due to the mode of operation of the monitor, *false positives* cannot occur: an execution time higher than the MID of a region, provided that WCETs are safely estimated, can only result from a control flow hijacking attack or a fault. On the other hand, attacks may stay undetected if they are fast enough (faster than the guaranteed attack detection latency).

Hardware constraints

Implementing the hardware monitor requires bounding the size of the Stack Memory (SM) and Region Memory (RM). Moreover, we want the monitor to analyze a PC value at every clock cycle. Bounding the resources required for implementing the monitor imposes the following constraints on the tree of selected regions:

- **Limited height** (maximal number of regions possibly active at the same time) to have a bounded size for SM
- **Limited arity.** This constraint allows to bound the number of simultaneous accesses to RM when several sub-regions (children in the region tree) can be entered from the currently active region.

The region selection algorithm presented in Section 3 assumes that the respective sizes of RM and SM, as well as the maximum tree arity are known. Selecting the best values for these parameters is achieved when dimensioning the hardware monitor for a given application. This impact of hardware constraints on attack detection latency is studied in Section 5.1.4.

3 Off-line selection of monitored regions

This section is devoted to a description of the algorithm we propose to select the regions to monitor. First, the properties of monitored regions are defined in Section 3.1. The algorithm, that guarantees a limited attack detection latency under the hardware constraints mentioned in Section 2 as well as a full program coverage is presented in Section 3.2, and an analysis of its complexity is sketched in Section 3.3. Moreover, we prove in Section 3.4 that, in the absence of hardware constraints, the algorithm is optimal (finds the smallest MAW).

3.1 Properties of monitored regions

Monitored regions are Single-Entry Single-Exit (SESE) regions (regions with a single entry edge and a single exit edge). They are extracted at compile time from the Control Flow Graph (CFG) of program binaries. For the scope of this paper, we use *canonical SESE regions* (simply called SESE regions in the following) as defined in [9]. A canonical SESE region (a, b) is by definition a SESE region such that no other SESE regions starting at the edge a (resp. ending at b) is included in (a, b). Canonical SESE regions can be viewed as the smallest SESE with a as entry edge or b as exit edge. By construction, the smallest size of a SESE region is a basic block (sequence of machine instructions with no branch except the last

instruction). SESE regions are proven in [9] to be properly nested (either completely nested or disjoint, at the function level), which allows them to be organized as an inter-procedural region tree. This is illustrated in the left part of Figure 3, where we can see a CFG with the SESE regions surrounding the nodes they contain. Region R_0 is disjoint from R_1. Regions R_3, R_4 and R_5 are nested in R_1. The corresponding tree structure is given in the right part of Figure 3.

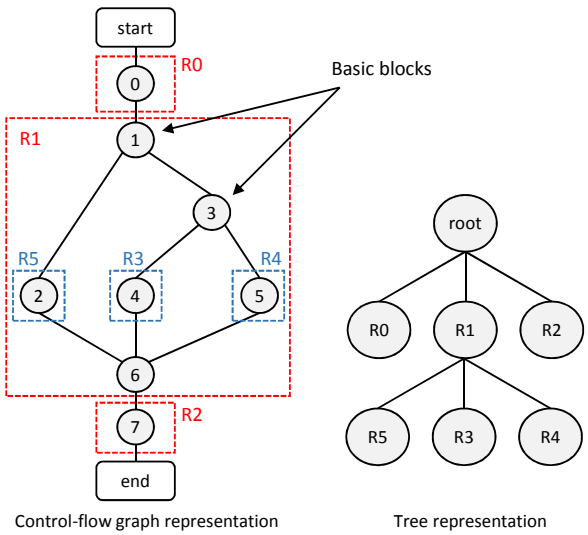


Figure 3 Single Entry Single Exit (SESE) regions viewed in the Control Flow Graph of the application and organized as a tree.

3.2 Region selection algorithm

According to the behavior of the monitor presented in Section 2, and assuming a set of monitored regions S, an attacker has at most the highest Maximal Inner Duration (MID) among monitored regions to perform their attack. As defined previously, the highest MID defines the guaranteed Maximum Attack Window (MAW).

The objective of the region selection problem is to select the monitored regions, that altogether must cover the entire program, such that their maximum MID is as small as possible. The selection operates under resource constraints (memory for stack and storage of region information, maximum number of regions entries/exists to check at every clock cycle). The region selection algorithm is iterative and greedy (never performs backtracking). At each iteration, the algorithm optimizes (reduces) the MID of the selected region R_m having the highest MID. The MID is reduced by selecting one sub-region S of R_m (children of R_m in the inter-procedural region tree), since by definition, other regions have no impact on the MID of R_m. The sub-region S is selected using a heuristic that estimates the new maximum attack window should S be selected. Finally, once S is selected, a new round of MID estimation takes place, with the new set of selected regions.

The algorithm is sketched in Listing 1. Variable *All* contains all regions (regardless of whether they will be selected or not). Variable *Selected* contains the current set of selected regions with their MID. R_m is the region under optimization and S the selected sub-region. Function *MaximalRegion* returns the region having the highest MID. Function *MIDEstimation(r,s)* computes (or re-computes) the MID of region r assuming a null execution

```
# Initialisation
for r in All:
  MIDEstimation(r,[]);
Rm = MaximalRegion(All);
Selected.Append(Rm);

# Main loop
while remainingSpace > 0:
  S = SelectSubRegion(Rm,Selected);
  if S == None:
    break
  Selected.Append(S);

  for r in All:
    if r in UpperRegions(S):
      MIDEstimation(r, Selected);

  Rm = MaximalRegion(Selected);
```

Listing 1 Region selection algorithm

time for the regions in set s. Finally, function *SelectSubRegion(r,s)* implements the heuristic and selects a sub-region of r given the current set of selected regions s. In Listing 1, the MID of all regions is first computed and the region that covers the entire program is selected. The main loop of the algorithm then iteratively optimizes the region R_m with the highest MID.

There are two ways the algorithm can terminate. The first possibility is that there are no more sub-regions to select. This happens if all regions in a program have been selected, or if the region with the highest MID does not contain any unselected sub-region. The second possibility results from an impossibility to find a sub-region that meets the hardware constraints. Infringement of hardware constraints is tested in the main loop through variable *remainingSpace* (test of limited space for region storage) and in function *SelectSubRegion* (test of limited storage for the region stack and limited arity for the region tree).

Function *SelectSubRegion(R,Selected)* implements the heuristic that calculates a *score* for each as-yet-unselected sub-region S of R. The score estimates the new maximum attack window should R be selected, and then selects the region R having the best (here lowest) score. The score is computed as follows:

$$score = max(MID(R, Selected) - T(R, Selected, S), MID(S, Selected))$$

This score represents the best-case evolution of the MAW, i.e., what the new MAW would be if there was no other path than the worst-case path in the region. $T(R, Selected, S)$ represents the contribution of sub-region S to MID(R, *Selected*). The first parameter of the *max* function estimates the new MID of R should its sub-region S be selected. The second term is the MID of sub-region S. The score is the maximum of the two parameters because the MAW is defined as the maximum of the MID of all selected regions. As the score is computed for every sub-regions of R_m, the algorithm likely selects a sub-region that is not a direct child of R_m.

After a sub-region S has been selected by the algorithm, all the MIDs affected by the newly selected region S have to be re-estimated. The affected MID are the ones of the regions that are higher in the inter-procedural region tree and thus include the selected region S.

3.3 Complexity of region selection

We first evaluate the worst-case complexity of the region selection algorithm in terms of the number of operations on the regions. At each iteration, the algorithm selects a sub-region R_m. Such a selection requires verifying the stack and arity constraints for each sub-region whose number is, at worst, the number of regions in the program, n. The stack constraint check consists in finding the longest weighted path in the inter-procedural region tree, whose depth is at worst the number of regions. We can thus infer that the overall worst-case complexity of the stack check is $O(n^2)$. Verifying the arity requires checking, for each region already selected inside the maximal region, whether the newly selected sub-region increases its arity beyond the constraint. The arity of a region can be computed with a slightly modified depth-first search. Thus the worst-case complexity of arity checking is $O(n^3)$. As the number of iterations of the algorithm is, in the worst-case, the number of regions, we obtain a worst-case complexity of $O(n^4)$ in terms of the number of operations on regions.

As presented in section 5, most of the run-time of the region selection algorithm is spent in MID estimation. The complexity of MID estimation is equivalent to the complexity of WCET estimation, and is hidden in the WCET estimation tool. Thus, the most important complexity metric is the worst-case number of MID estimations. This number can be bounded as, in the worst-case, the algorithm evaluates the MID of all regions. Thus, the number of MID estimations cannot exceed n^2. In practice, the actual number is far lower, as the algorithm only has to estimate the MID of the regions higher in the inter-procedural region tree than the newly selected region, as the MID of other regions is not impacted. The observed run-time of region selection is studied in Section 5.1.2.

3.4 Maximum Attack Window optimality: proof sketch

The region selection algorithm is an iterative greedy algorithm, which selects a set of regions to monitor under hardware constraints (see Section 2) and aims at minimizing the MAW. The proposed algorithm satisfies the following optimality property: if hardware constraints are ignored, our algorithm selects the regions that minimize the MAW (as if *all* regions were selected). Proving this property relies on two lemmas.

▶ **Lemma 1.** *When a new region is added to the set of selected regions, the MAW cannot increase.*

▶ **Lemma 2.** *If adding a new region reduces the MAW of an application, then it is a sub-region of the selected region having the maximal MID.*

Only a proof sketch is given for Lemmas 1 and 2 for space considerations. Lemma 1 can be proven by noting that if a region R has a MID m, then any path within this region has a MID lower than or equal to m. Thus selecting a sub-region S inside R only lowers or maintains the time on those paths, and thus the MID with S selected, and consequently the MAW, does not increase. Lemma 2 is proven using the idea that, as regions are properly nested, if a selected region S impacts the MID of another region R, then S must be nested into R to decrease the time of some instructions in at least one path of the affected region.

Assuming Lemmas 1 and 2 hold, let us show that the algorithm selects the regions that minimize the MAW when the algorithm stops. According to the code of the algorithm, there are three possible causes for the algorithm to end:

1. Hardware constraints are not met (lines 8 and 9 in the algorithm). This cannot happen here because we ignore hardware constraints.

2. All regions have been selected. Since according to Lemma 1 the MAW never increases when selecting a new region, we have then reached the minimal MAW.
3. The region having the maximal MID has no as-yet-unselected sub-regions. Then no region could reduce the MAW according to Lemma 2.

Consequently, when the algorithm ends, no region could further reduce the MAW. ◀

4 Implementation

4.1 Target processor and compilation toolchain

Without loss of generality, we target in this paper a Leon 3 core, implementing the SPARC V8 instruction set, and in which we have deactivated the instruction and data caches. Programs are compiled using the *gaisler* compiler tool-chain [5]. The Leon 3 processor has *branch delay-slots*: the instruction located immediately after a (conditional or unconditional) branch instruction will always execute, regardless of the outcome of the branch. The processor also includes a branch predictor.

WCET and MID estimations use the commercial WCET estimation tool aiT for Leon 3, version 19.04i [2]. aiT implements value analysis using abstract interpretation, to determine the range of values in registers, automatic detection of loop bounds [6], as well as static analysis of the processor micro-architecture. This version implements a modification to the aiT original behavior to prevent the timing analysis of a part of the code (the time is then given by the user); the modification maintains the value analysis but discards the micro-architectural analysis.

4.2 Implementation of region selection

Region selection has been implemented in Python3. In a first step, the Control-Flow Graph is constructed, using the disassembled code of the provided binary. SESE regions are then extracted from the CFG, using the algorithm presented in [9]. Finally, we perform the region selection as previously presented in Listing 1. MID estimations are performed using aiT, by computing the WCET of the targeted regions and using aiT annotations to set the WCET of selected sub-regions to *zero*.

The algorithm computing the SESE regions provides the smallest SESE regions, with the minimum region size of one basic-block. Some of these regions are consecutive and thus can be merged into a larger SESE region, called *domain*. Including domains in the program region tree allows to perform the selection of larger regions with less resources. For example, instead of selecting 4 regions composing a loop body, it is possible to select the domain representing the fusion of these 4 regions, reducing memory requirements and mitigating arity constraints. It is still possible to select one of the 4 regions inside the domain to further reduce the MAW.

4.3 Implementation of the hardware monitor

As described in Section 2, the monitor needs to analyze a new PC value at each cycle (that may be the same as the one observed at the previous cycle). Based on this observed value, it decides whether to exit the active region and/or to enter a new one. In the following, we describe the design of the monitor capable of handling 4096 regions, with a maximal arity of 8 and a stack size of 64.

The program region tree is stored in the Region Memory (RM) using an array with 64 bits per region. Those 64 bits are divided in the following way:

- 24 bits for the address of the entry point (target of the region's entry edge),
- 24 bits for the address of the exit point (target of region's exit edge),
- 12 bits for encoding the index of the first child in the array (index start with 0), and
- 4 bits for encoding the number of children.

Moreover, we ensure that all children of a given region are stored consecutively. Hence the index of the first child of a region gives access to all children. Figure 4 is an example of representations. We can see that all children of region R_1 are stored consecutively, starting at index 3. In order to save memory, the MID of every region is not stored in RM. Instead, we use the MAW (highest MID) for all regions. This means that attacks are detected only when the counter of the current region reaches the MAW and not earlier (for regions whose MID is lower than the MAW). On the other hand, not saving the MID for every region saves memory and allows us to monitor more regions. RM is divided into eight banks, a region of index i being stored in bank i *mod* 8. Consequently, all children of a region can be accessed within a single clock cycle.

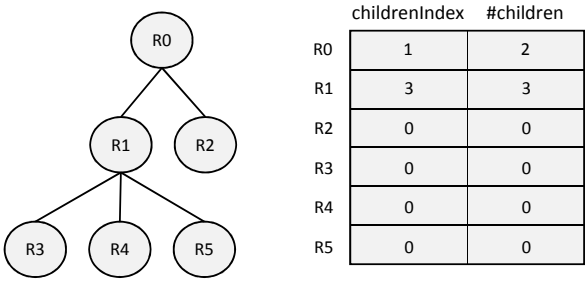


Figure 4 Example of the tree encoding used in the monitor. For the sake of simplicity, addresses of entry and exit points of each regions are not depicted in the figure.

The Stack Memory (SM) is encoded using 64-bit registers. In each register, 24 bits are used to store the counter value. The remaining bits contain a copy of the information on the region (24 bits for region exit, 12 bits for first children, 4 bits for the number of children) such that the monitor can test region exit without any access to RM.

The hardware monitor is described at the C level and synthesized into hardware description language using Mentor Catapult HLS (v10.3a). The system has been simulated at the C level and is currently being implemented on an Altera FPGA.

5 Experimental evaluation

In this section, we present the experimental study we conducted to evaluate our approach. This study is separated into two parts.

- Section 5.1 is dedicated to an evaluation of the region selection algorithm. We first study the effectiveness of region selection and the limits in the way we build regions. Then we measure the execution time of the selection algorithm, ignoring all hardware constraints. Finally, we evaluate the impact of hardware constraints on the quality of the results (MAW).
- Section 5.2 evaluates the hardware monitor. We first measure the latency of the attack detection. Then we compare our hardware-based approach against existing software-based

methods. Finally, we evaluate the silicon area required to implement our approach and compare it with the area of the Leon core.

5.1 Evaluation of the region selection algorithm

5.1.1 Experimental setup

We use the Mälardalen [8] and PolyBench [17] benchmarks to evaluate the region selection algorithm. The Mälardalen benchmarks are composed of multiple programs written in C with many different program structures. In particular, among the benchmarks there are control-oriented programs such as *statemate* and *nsichneu* and more simple computational kernels such as *fft1* or *fir*. The PolyBench benchmarks are computation kernels written in C, composed mostly of loop nests. To evaluate the region selection algorithm on as realistic as possible codes, we ruled out all benchmarks with less than 30 regions in the program structure tree. Furthermore, we eliminated benchmarks that could not be analyzed with only basic loop bound annotations.

We evaluate our technique on the following 24 programs: 16 from the Mälardalen benchmarks: *adpcm*, *cnt*, *compress*, *crc*, *fft1*, *ludcmp*, *matmult*, *minver*, *ndes*, *nsichneu*, *statemate*, *ud*, *edn*, *st*, *lms* and *qsort-exam*, and 8 from the Polybench benchmarks: *gemver*, *3mm*, *ludcmp*, *covariance*, *nussinov*, *adi*, *fdtd-2d*, *heat-3d*. There are two implementations of the *ludcmp* algorithm, one in each benchmark suite. To differentiate them in the next section we add the prefix *poly_* to every program of the PolyBench suite.

5.1.2 Maximum Attack Window (MAW) without hardware constraints

We first estimate the best (i.e. smallest) MAW, obtainable by the selection algorithm when ignoring hardware constraints (size/arity). In Table 1, we compare, for each benchmark, the obtained MAW (obtained by the selection algorithm) with the WCET of the benchmark (equivalent to the MAW at the start of region selection). We also indicate the total number of regions per benchmark and the number of regions selected by the algorithm. The results show that 60% of the benchmarks have a MAW below 820 cycles. On the selected architecture, an instruction takes about 12 cycles (counting the SRAM access time). Thus, for 60% of the benchmarks, the MAW is about 69 instructions.

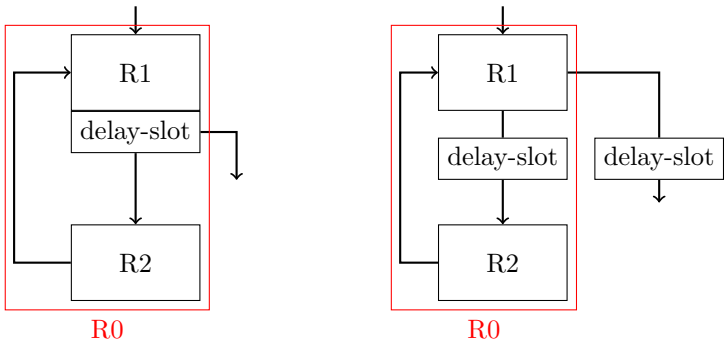
For 14 out of 24 benchmarks, the obtained MAW corresponds to a region containing a single basic block (labeled with a 'b' in the "Limiting reason" column). For these benchmarks, the MAW is between 274 and 5 039 cycles, with 50% of these benchmarks having a MAW below 614 cycles.

For the other 10 benchmarks (labeled with an 's'), higher values of MAW are obtained due regions containing a loop with a delay-slot. *aiT* handles delay-slots as special basic blocks to handle branch prediction. As the impact of a delay-slot on the WCET may differ depending on whether the branch is taken or not taken, *aiT* considers the delay-slot as two basic blocks, one for each outcome. This prevents us from placing the delay-slot instruction as the last instruction of a region. Thus, delay-slot instructions are often placed in an upper region in the region tree. As loops almost always have a header with a delay-slot, the delay-slot is placed in the region representing the loop. As the loop can iterate many times, the cost of the delay slot is multiplied by the loop bound, resulting in a region containing only a single delay-slot instruction but with a high MID. This region can then prevent the improvement of the MAW if it becomes the region with the highest MID.

For example, we sketch the desired behavior in Figure 5a: the delay-slot is part of the region R_1 and thus the delay-slot execution time is counted once in the MID of R_1. Figure

Benchmark	WCET	MAW	overall regions	selected regions	Limiting reason
lms	9 404 429	1 609	76	35	s
qsort-exam	220 542	614	45	25	b
edn	1 774 300	3 155	78	32	b
st	3 218 793	8 001	60	18	s
poly_ludcmp	1 300 575 626	961	43	31	s
poly_heat-3d	916 894 881	1 953	35	9	b
poly_gemver	20 264 113	961	33	22	s
poly_nussinov	3 906 470 200	493	39	21	b
poly_3mm	254 765 399	450	44	24	b
poly_adi	552 919 400	1 456	33	18	b
poly_fdtd-2d	289 221 703	1 346	38	19	s
poly_covariance	379 138 134	801	34	21	s
ndes	1 225 593	661	101	38	b
matmult	3 339 921	340	53	24	b
compress	1 758 363	166 358	102	79	s
statemate	144 066	2 970	362	21	b
ludcmp	113 587	421	64	29	b
adpcm	2 116 152	16 001	175	17	s
fft	439 514	1 117	75	17	b
minver	35 614	5 039	62	8	b
crc	764 776	4 355	33	16	s
ud	103 136	398	56	27	b
nsichneu	231 282	21 891	755	753	s
cnt	79 166	274	36	17	b

Table 1 Best attainable MAW, obtained when ignoring hardware constraints. For each benchmark, we provide the WCET of the whole application, the MAW obtained, the total number of regions and the number of selected regions. The last column gives the limiting factor for MAW reduction: either the size of the basic block (labeled 'b') or the way delay slots are handled (labeled 's').



(a) Desired behavior **(b)** Actual behavior

Figure 5 Delay-slot handling example

5b depicts the behavior of *aiT*, which duplicates the delay-slot for each branch outcome. It also adds guards on the delay-slot block, which forces us to include its execution time in R_0. As R_0 contains the loop, the MID of R_0 depends on the number of iterations. This situation occurs in the *st* benchmark, which contains a 1 000-iterations loop and a delay-slot whose WCET is estimated at 8 cycles. This results in an indivisible region of 8 001 cycles that blocks the improvement to the MAW.

5.1.3 Run-time of region selection without hardware constraints

The run-time of region selection, when ignoring hardware constraints, is given in Figure 6. Each benchmark is represented by a bar with the total time of MID estimations (*aiT* runtime) as one part (top) of the bar and the run-time of the rest of the region selection algorithm as the other part (bottom). We do not represent *nsichneu* as it would have flatten all the other benchmarks but we give the different times.

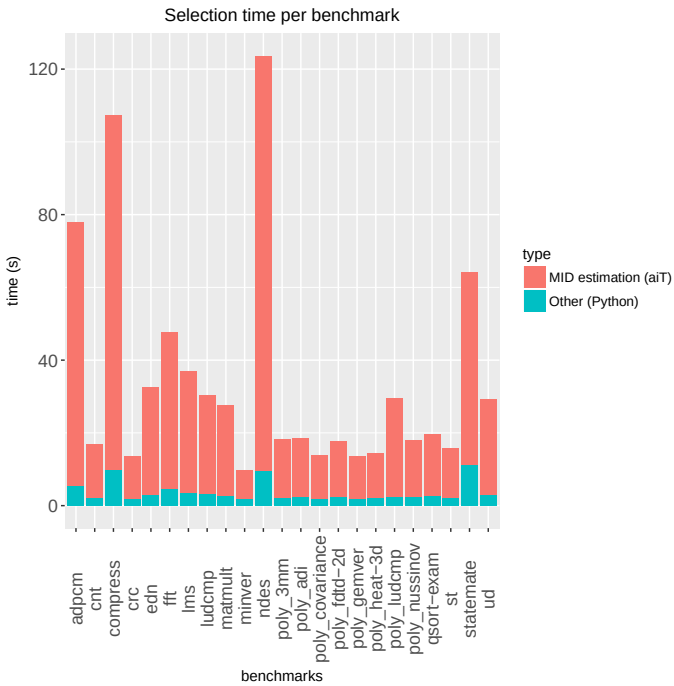


Figure 6 Run-time of the region selection algorithm. *nsichneu* is not presented to be able to avoid flattening the other benchmarks. For *nsichneu*, MID estimations (aiT) = 4 440 s, Other (Python) = 317 s

These results empirically show that most of the computation time of the selection algorithm comes from the MID estimations by *aiT*. We also observe that the overall execution time remains acceptable for all represented benchmarks. In all benchmarks except *nsichneu*, region selection is performed in less than 200s. *nsichneu* takes more time due to its structure that forces the selection algorithm to select almost all the regions, which requires many MID estimations. Most programs do not have a structure as specific as the one of *nsichneu* and for them, the selection algorithm run-time remains very low.

5.1.4 Impact of hardware constraints

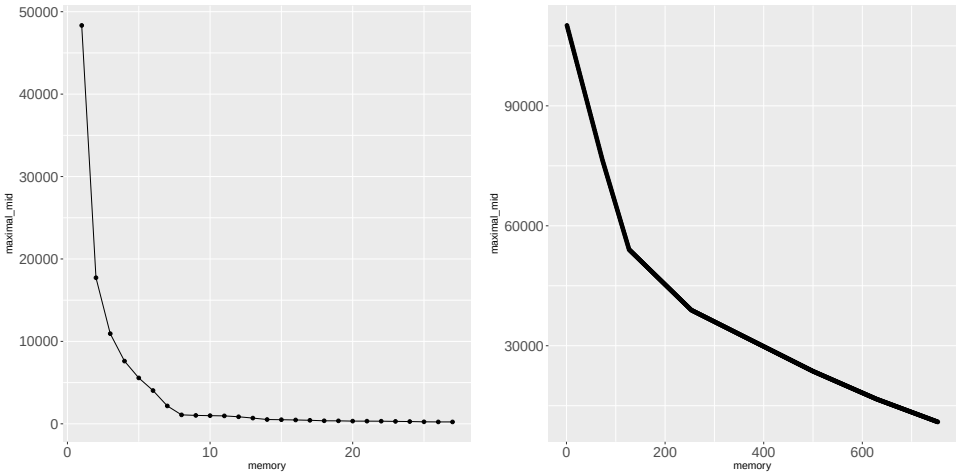
This section studies the impact of hardware constraints on the quality of region selection.

First, we observed that the required stack size in the benchmarks never exceeds 12 and the required size per stack element is 64 bits only. Thus, this factor has little importance on the resources consumed by the monitor. Consequently, we limit our analysis to the impact of limited memory for RM and limited arity for the region tree.

We first evaluate how limited size for RM impacts the obtained MAW. To do so, we extract the MAW for each iteration of the region selection algorithm without constraints. We can then have a representation of the MAW obtained if the selection algorithm had stopped due to limited RM size, without other constraints (stack / arity). The obtained curves depict the obtained MAW as a function of the number of selected regions, with each point being a newly selected region (which is equivalent to an increase of the RM size by 64 bits). Most of the curves have the same shape as *ud*, presented in Figure 7a.

For most benchmarks, the threshold plummets at the beginning of the selection and then decreases slowly or stabilizes, corresponding to two phases. The only benchmark for which the MAW decreases at approximately a constant rate is *nsichneu* as depicted in Figure 7b. This is due to the particular structure of this benchmark, imposing a one-by-one selection of a very small region at each iteration. Interestingly, on this benchmark, the algorithm does select from the most to the less interesting regions, as the slope of the curve decreases with the number of regions selected.

The best attainable MAW is reached when selecting a mean of 47% of the overall regions, with a standard deviation of 20% (noted 47±20%). When selecting the regions, 90% of the reduction is performed by selecting only 24%±19% of the regions selected at the end. The first selected regions are often very interesting to efficiently reduce the MAW. The large standard deviations are mostly due to the disparity in the complexities of the benchmarks.



(a) ud **(b)** nsichneu

Figure 7 MAW evolution with the number of selected regions

The second studied hardware constraint is the arity constraint. To evaluate its influence, we perform the selection in three configurations, all without constraining the size of RM: unbounded arity, arity of 8 and arity of 4. The resulting MAW is given in Table 2.

The results indicate that for an arity of 8, all benchmarks except *nsichneu* have a MAW

less than 2 times higher than with no arity constraint. 21 out of the 24 benchmarks have the same MAW as when the arity is unbounded. *nsichneu* remains a special case due to its structure, which makes the selection rely exclusively on the arity to reduce the MAW. We see in that case that the structure of the code has a strong impact on the capacity of our method to protect it.

For an arity of 4, 16 out of 24 benchmarks are not affected by the constraint. Among the affected 8 benchmarks, the increase in the MAW is so low that there is almost no difference in the protection for 3 of them. However, the other benchmarks except *compress*, see their MAW increase by a factor of 6 or more compared to a system with no arity constraint. In these cases, the attack window may be considered too large to efficiently protect the program and a superior arity is then required. *compress* is special as it has a very high MAW to begin with, so even a factor below two makes its MAW skyrocket even more. We conclude that arity has an important impact on the obtained MAW, but it seems that there is no need to increase the arity to more than 8 to have good results for almost all benchmarks.

Benchmark	**No bound**	**4**	**8**	**Benchmark**	**No bound**	**4**	**8**
adpcm	16 001	16 001	16 001	cnt	274	274	274
compress	166 358	313 124	196 406	crc	4 355	4 355	4 355
edn	3 155	145 340	3 155	fft	1 117	1 117	1 117
lms	1 609	1 609	1 609	ludcmp	421	621	421
matmult	340	340	340	minver	5 039	5 039	5 039
ndes	661	707	661	nsichneu	21 891	227 184	223 200
poly_3mm	450	450	450	poly_adi	1 456	1 456	1 456
poly_covariance	801	801	801	poly_fdtd-2d	1 346	1 346	1 346
poly_gemver	961	961	961	poly_heat-3d	1 953	1 953	1 953
poly_ludcmp	961	961	961	poly_nussinov	493	518	493
qsort-exam	614	4 047	1 262	st	8 001	8 001	8 001
statemate	2 970	3 014	2 970	ud	398	398	398

Table 2 MAW in function of arity for all benchmarks (no limit on RM size)

5.2 Evaluation of the hardware monitor

This section evaluates the hardware monitor, in terms of observed detection latency and implementation overheads.

5.2.1 Observed attack detection latency

The experiments reported in this section have two objectives: to demonstrate that the proposed approach effectively detects control-flow deviations and evaluate the achieved attack detection latency.

The experiments operate on traces obtained using *Modelsim SE-64 10.5a*, that performs a cycle-accurate simulation of the Leon 3 core while it executes code. We post-processed the traces produce by Modelsim to extract the trace of PC values at each cycle and remove the C values corresponding to the instructions canceled after branch misspredictions. The C version of our monitor is configured with the result of our region selection algorithm with stack and memory constraints set at 1024 and arity constraint set at 8 as for our hardware evaluation (see section 5.2.3). It scans the trace to detects when the execution enters or

exits. We simulate a scenario where the attacker escapes from the standard execution path from an unknown path and never returns back. This is done by modifying the PC value in the execution trace at a cycle picked randomly. Then we verify that the proposed hardware monitor detects the attack, and we measure the latency between the attack and its detection. For each application, the attack is inserted at 100 000 random points. Due to experimental limitations, some optimizations on the estimation of the MAW are disabled for this experiment. Consequently, the MAW used here are different from the MAW presented in Table 1. Results of this experiment are presented in Table 3, giving the mean and standard deviation of the attack latency, expressed as a percentage of the MAW.

Benchmark	**Mean latency** (% of MAW)	**Standard deviation latency** (% of MAW)
crc	99 %	0.8 %
lms	94 %	5.7 %
minver	68 %	29.8 %
fft	94 %	4.9 %
qsort-exam	97 %	2.7 %

Table 3 Latency of the detection of an attack using the proposed mechanism. The number of cycles between the attack and its detection are provided as a percentage of the MAW determined statically.

We observe that all attacks are detected by the monitor. The latency is often close to the MAW as the attacks often take place in regions with a MID far lower than the MAW. Consequently, even if the attack is triggered close to the end of the region, the alarm is raised when the counter reaches the MAW. The *minver* application has a different behavior: most of the execution takes place in the region having the limiting MID. Thus, when the attack is triggered at the end of the region, it is detected quickly. We can see in Table 3 that *minver* has a latency of 64% of its maximal MAW.

5.2.2 Evaluation of overheads as compared to software approaches

As we opted for a hardware implementation of the monitor, our approach has no impact on the execution time. In comparison, software-based approaches require to read a cycle counter from memory and to compare the value with the MAW. Depending on the implementation, this software implemented monitoring system could even require a system call, as in the work from Zimmer at al. [27].

In this experiment, we evaluate the overhead of our technique if it were implemented in software. We use the execution traces extracted during the previous experiment to count the number of times a region is entered or exited during the execution. Then, we define three scenarios with different penalties for switching of region (entry / exit), corresponding to different implementation strategies for the monitoring system (e.g. with/without system calls, optimized or not). In the *fast-soft* scenario, monitoring a region requires 10 cycles at each region switch. In the *medium-slow*, it requires 20 cycles and in the *slow-soft* it requires 200 cycles.

Table 4 summarizes the results of this experiment. The first column provides for each application simulated, the number of times the monitor switches from one region to another. The other columns correspond to the performance penalty due to the software implemented monitoring of regions. We provide both the number of cycles spent in monitoring and the

		Overhead (cycle / % of execution time)			
Benchmark	**Region switch**	**0**/switch (hardware)	**10**/switch (fast-soft)	**20**/switch (medium-soft)	**200**/switch (slow-soft)
crc	1 718	0	17 180 19%	34 360 39%	343 600 387%
lms	6 134		61 340 5%	122 680 10%	1 226 800 99%
minver	12		120 1%	240 3%	2 400 29%
fft	1 090		10 900 28%	21 800 56%	218 000 556%
qsort-exam	306		3 060 51%	6 120 103%	61 200 1029%

Table 4 Comparison between the overheads of the hardware monitor (hardware) and the state of art assumed with a constant overhead at each region switch (entry or exit) in the worst-case scenario for a subset of the benchmarks

overhead relatively to the original execution time of the application. We can see that a software based approach can add a significant overhead in the execution time. Even for the *fast-soft* scenario, the overhead reaches 51% for *qsort-exam*. For *medium-soft* and *slow-soft*, the monitoring overhead is often larger than the original execution time. Of course, this overhead could be reduced by increasing the size of the monitored regions, but this also increases the detection latency. The hardware monitors does not affect the execution time of the application.

5.2.3 Hardware overhead of monitor

The last experiments estimate the hardware cost of the monitor. We have synthesized both the Leon core and the monitor, targeting a 28 nm technology from STMicroelectronics and a working frequency of 50 MHz (default frequency of the Leon core). As the cost is dominated by memory, this cost is separated from the other costs when giving the results.

Leon Memory Size	64 kB	128 kB	256 kB	512 kB
Monitor Size	8 kB (1 024 monitored regions)			
Leon Area (μm^2)	128 197	229 117	430 957	834 637
core only	27 277			
memory only	100 920	201 840	403 680	807 360
Monitor Area (μm^2)	28 598			
monitor only	15 983			
memory only	12 615			
Overhead	22%	12%	7%	3%

Table 5 Area cost of the Leon core and of the proposed hardware monitor. Results are given for different core configurations, with memory ranging from 64 kB to 512 kB, and a monitor capable of monitoring 1 024 regions (which corresponds to 8 kB).

Results are reported in Table 5. The first two lines are a summary of the configurations studied. Then, area results for the Leon core and for the proposed monitor (in µm2) are

presented. For both of them, we provide the area consumed ignoring the memories and the area of the memories. The last line gives the area overhead due to the use of the proposed approach. We can see that, depending on the core configuration, the overhead of the monitor goes from 22% to 3%. All these results corresponds to a monitor handling up to 1 024 regions. This value was picked knowing that the biggest application we studied requires 80 regions (ignoring *nsichneu* that requires 700 regions). We can estimate that doubling the number of region handled by the monitor would double the area consumed by the memories.

6 Discussion

As any security mechanism, the design we propose has a limited scope that we discuss in this section. Since our work heavily relies on WCET estimation techniques and tools, their limitations impact the quality of our work. In particular, the pessimism of WCET estimates, even on small regions such as basic blocks, induces higher bounds for our detection and thus more time for the attacker to bend the program behavior.

We selected an implementation of the monitor that uses the same bound for all regions, to reduce the hardware overhead of the design. Storing the bound of each region in the memory and checking the elapsed time against this bound would certainly improve the security of the protection, at the expense of a higher hardware overhead to store all these data.

Using debug interfaces already present on the processor is an interesting idea to use this protection with less hardware overhead. For example, we could implement the monitor in software and use another core to run it in order to reduce the intrusiveness). Extension to debug-interface is left to future work. Another way to improve the protection would be to use less pessimistic WCET estimations (e.g. using hybrid methods) but this may provide unsound WCET estimates which would detect attacks where there are not (i.e. false-positive).

A second limitation is the type of attacks that can be detected by our protection. Our protection targets attacks that would modify the control-flow and not return at the right place in time. Our design does not try to prevent attacks that are not based on control-flow hijacking (e.g. data-only attacks [11]).

Finally, our current design does not yet handle multi-tasking operating system and multi-processor. We believe that multi-processors can be easily handled by dispatching the task on the processors off-line and using one monitor per processor. Multi-tasking operating systems on the other hand requires to detect context switches such that it also switches the monitor context. It also requires to partitioning the memory of the monitor to maintain the context of suspended tasks while running another task. To ensure that the right task is being monitored, the protection must also protect the scheduling. Finally, the protection have to protect the operating system itself. This is left to future work.

7 Related work

Many methods exist to protect systems, including real-time systems, against attacks. A first class of techniques is to *prevent* the attacks from happening, for instance by hardening the binary using checksums, or by modifying the memory layout of programs, or by introducing some randomness when generating schedules. Address Space Layout Randomization (ALSR) randomizes the start address of the key memory regions of a process (code, data, stack, libraries) to guard against buffer overflows. The use of ASLR in small embedded systems (for example, on 8 or 16-bits architectures) is less effective than in 32 or 64-bits architectures, as the system does not always contain enough entropy for them to be efficient [18]. In addition,

when timing guarantees are required, WCET estimation techniques have to be re-designed for supporting the presence of unknown addresses for memory regions. The research presented in [7] proposes WCET estimation techniques (in particular instruction cache analysis) for different diversification granularities. This work was able to compute a far tighter WCET estimate than the one obtained with *all miss* assumption (i.e., equivalent to a system without a cache). However, a rough analysis of the data provided in this paper shows that the cost of fine grain diversification can be up to a 50% worst-case overhead, which is often too much for industrial purposes [21]. Another level at which diversification can be used in real-time systems is at the schedule level. In [10], the goal is to prevent targeted side-channel attacks on other tasks by *shuffling* the schedule while conserving the deadlines criteria, either off-line or on-line.

A second class of techniques, complementary to prevention, is to let the attacker perform some actions and detect them before they cause harm to the system. *Detection techniques* use monitoring techniques, implemented either in hardware or in software, to look for the results of the attack, and not the causes. The sensitivity of the monitor is used to trade-off the security the monitor provides against how much it interferes with the system. The technique proposed in this paper is a detection technique.

Detection techniques differ by the class of information they monitor.Yoon et al. [26] monitors system calls at run-time and compares them to a list of system calls under normal execution. Walls et al. [23] presents a software technique that checks the integrity of the control flow of a program by monitoring the target of indirect branches and comparing them to the value for a normal execution. Hardware for control flow integrity verification is presented in [1]. Zimmer et al. [27] monitors execution times of code snippets of different granularities and compares them against estimated WCETs, to detect attacks such as control-flow hijacking. Like [27] we use timing information for attack detection. However, our research differs from [27] in several aspects: (i) we use dedicated hardware instead of software for detection of timing violations, and then have no impact on the predictability of monitored programs; (ii) we propose an algorithm to automate the selection of monitored regions and thus provide guarantees on the attack detection latency. In addition, in comparison to [27], we do not need Best-Case Execution Times (BCET) in our approach, a metric that is not currently supported by state-of-the-art WCET estimation tools.

Watchdogs have been intensively studied to protect against faults in RTES [12] [25]. In [12], the authors present a survey of different techniques to protect the control flow and memory accesses. The control-flow protection uses signatures inserted at the beginning of each protected block. These signatures are read by the watchdog which can ensure that the block corresponds to the signature and that the received signatures follow a correct order. In [25], the authors use the WCET of a block as a signature to ensure the correct timing behavior of the program. However, it does not handle computed branches. All these techniques require an instrumentation of the program that slows down the program while our solution can be applied without modification of the program. The second point is that these techniques are not focused on security. A fault happening naturally has a very low probability of modifying the signature or modifying the instructions while maintaining a correct signature. On the other hand, a well-crafted attack could subvert the system by modifying the time allocated for a block and transmitted to the watchdog.

A crucial characteristic of a monitor — that impacts its security, responsiveness and invasiveness — is if the monitor is made on *dedicated hardware* or in *software* by the system. Hardware-based monitors have better responsiveness as they can monitor the system faster than software can. As they use dedicated hardware, this kind of monitor uses less

system resources, thus reducing the invasiveness of the monitor and reducing its impact on the determinism. Finally, the security is reinforced because hardware is dedicated to the monitoring and can thus be isolated from the system as well as protect the system itself. However, these advantages come with the cost of developing dedicated hardware for each type of required monitor [20] [16] or using a dedicated co-processor [4]. The monitor proposed in this paper is fully implemented in hardware, and is shown not to impact the predictability of software. Protection of the monitor itself against attacks, although not addressed in the current status of our work, should be facilitated by the limited and well-delimited amount of information used for the monitor operation.

8 Conclusion and future work

We have presented in this paper a technique to detect attacks in real-time systems based on WCET of code regions. A monitor, implemented in hardware, tests if a code region exceeds its WCET a the result of an attack, and raises an alarm should this happen. Experimental results show that the Maximum Attack Window guaranteed by our approach is as short as a basic block (tens of instructions for most codes).

Two factors limit the MAW guaranteed by our approach. The first one is the presence of delay-slots in the targeted architecture. Delay-slots, as detailed in Section 5, artificially enlarge the MAW, because they are accounted for in the enclosing selected region. This issue will not appear on architectures without delay-slots, but unfortunately would need a deep change in aiT to have a more precise estimation of the MAW. The other factor that prevents an extremely small MAW is the presence of long basic blocks in some applications. Addressing this issue would require to split long basic blocks in several regions, which would add complexity to the region selection algorithm. A promising direction would be to subtly change the algorithm as follows: in case the MAW is the duration of one basic block the basic block is split in two equal-length (in terms of number of instructions) regions.

Another way to improve the efficiency of attack detection would be to use *Best Case Execution Times* (BCET) in addition to WCETs. We did not further follow this direction for technical reasons, because state-of-the-art WCET estimation tools are strictly oriented towards worst-case performance and do not provide BCET values.

For the scope of this paper, we addressed the monitoring of a single task on a mono-processor system. Extending this work for multi-processors and multi-tasks operating systems is left for future work.

As for the hardware implementation of the monitor, the monitor is currently described as C code and synthesized into VHDL using using the High Level Synthesis (HLS) tool Catapult. Functional correction has been demonstrated through simulations. Ongoing work concerns the integration of the monitor on an FPGA board embedding the Leon 3 core. We also plan as future work to protect the monitor itself, in particular its memory, against fault attacks, using redundancy techniques. The monitor will then be tested against real attacks, including fault attacks, and the observed attack detection latencies will be observed and compared to the guaranteed latency.

9 Acknowledgements

We would like to thanks Steven Derrien for the discussions that lead to this research and Stefanos Skalistis for his insight. We also warmly thank AbsInt for providing the aiT WCET estimator and modifying it for meeting our needs for region selection.

References

1. F. A. T. Abad, J. V. D. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan. On-chip control flow integrity check for real time embedded systems. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 26–31, Aug 2013. doi:10.1109/CPSNA.2013.6614242.
2. AbsInt GmbH. ait worst-case execution time estimation tool. https://www.absint.com/ait/. Last accessed: 2020/01/22.
3. Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. A Novel Side-Channel in Real-Time Schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 90–102, April 2019. ISSN: 1545-3421. doi:10.1109/RTAS.2019.00016.
4. Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. Co-processor-based behavior monitoring: Application to the detection of attacks against the system management mode. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*, pages 399–411, 2017. URL: https://doi.org/10.1145/3134600.3134622, doi:10.1145/3134600.3134622.
5. Cobham Gaisler. Compiler toolchain for the leon processor. https://www.gaisler.com/. Last accessed: 2020/01/22.
6. Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2007/1193, doi:10.4230/OASIcs.WCET.2007.1193.
7. Joachim Fellmuth, Thomas Göthel, and Sabine Glesner. Instruction caches in static WCET analysis of artificially diversified software. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, pages 21:1–21:23, 2018. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2018.21, doi:10.4230/LIPIcs.ECRTS.2018.21.
8. Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010. URL: http://www.es.mdh.se/publications/1895-.
9. Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. *SIGPLAN Not.*, 29(6):171–185, June 1994. URL: http://doi.acm.org/10.1145/773473.178258, doi:10.1145/773473.178258.
10. Kristin Krüger, Marcus Völp, and Gerhard Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, pages 22:1–22:17, 2018. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2018.22, doi:10.4230/LIPIcs.ECRTS.2018.22.
11. Tingting Lu and Junfeng Wang. Data-flow bending: On the effectiveness of data-flow integrity. *Computers & Security*, 84:365–375, July 2019. URL: http://www.sciencedirect.com/science/article/pii/S0167404818312604, doi:10.1016/j.cose.2019.04.002.
12. A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors—a survey. *IEEE Transactions on Computers*, 37(2):160–174, February 1988. doi:10.1109/12.2145.
13. Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.
14. L. O. Murchu N. Falliere and E. Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5:6, 2011.
15. Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: hacking tesla from wireless to can bus. *Briefing, Black Hat USA*, pages 1–16, 2017.

16 C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. Tainthls: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018. doi:10.1109/TCAD.2018.2834421.

17 Louis-Noël Pouchet and Tomofumi Yuki. PolyBench/C. URL: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

18 Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1030083.1030124, doi:10.1145/1030083.1030124.

19 Peter Silberman and Richard Johnson. A comparison of buffer overflow prevention implementations and weaknesses. *IDEFENSE, August*, 2004.

20 C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, May 2016. doi:10.1109/SP.2016.9.

21 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 48–62, 2013. URL: https://doi.org/10.1109/SP.2013.13, doi:10.1109/SP.2013.13.

22 N. Timmers, A. Spruyt, and M. Witteman. Controlling PC on ARM using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35, Aug 2016. doi:10.1109/FDTC.2016.18.

23 Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany.*, pages 2:1–2:24, 2019. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2019.2, doi:10.4230/LIPIcs.ECRTS.2019.2.

24 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. URL: https://doi.org/10.1145/1347375.1347389, doi:10.1145/1347375.1347389.

25 Julian Wolf, Bernhard Fechner, Sascha Uhrig, and Theo Ungerer. Fine-grained timing and control flow error checking for hard real-time task execution. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 257–266, June 2012. ISSN: 2150-3117. doi:10.1109/SIES.2012.6356592.

26 Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, IoTDI 2017, Pittsburgh, PA, USA, April 18-21, 2017*, pages 191–196, 2017. URL: https://doi.org/10.1145/3054977.3054999, doi:10.1145/3054977.3054999.

27 Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. Time-based intrusion detection in cyber-physical systems. In *ACM/IEEE 1st International Conference on Cyber-Physical Systems, ICCPS '10, Stockholm, Sweden, April 12-15, 2010*, pages 109–118, 2010. URL: https://doi.org/10.1145/1795194.1795210, doi:10.1145/1795194.1795210.