RT-DFI: Optimizing Data-Flow Integrity for Real-Time Systems

Nicolas Bellec, Guillaume Hiet, Simon Rokicki, Frédéric Tronel, Isabelle Puaut

RT-DFI: Optimizing Data-Flow Integrity for Real-Time Systems

Nicolas Bellec
Univ Rennes, Inria, CNRS, IRISA, France

Guillaume Hiet
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, France

Simon Rokicki
Univ Rennes, Inria, CNRS, IRISA, France

Frederic Tronel
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, France

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA, France

Abstract

The emergence of Real-Time Systems with increased connections to their environment has led to a greater demand in security for these systems. Memory corruption attacks, which modify the memory to trigger unexpected executions, are a significant threat against applications written in low-level languages. Data-Flow Integrity (DFI) is a protection that verifies that only a trusted source has written any loaded data. The overhead of such a security mechanism remains a major issue that limits its adoption. This article presents RT-DFI, a new approach that optimizes Data-Flow Integrity to reduce its overhead on the Worst-Case Execution Time. We model the number and order of the checks and use an Integer Linear Programming solver to optimize the protection on the Worst-Case Execution Path. Our approach protects the program against many memory-corruption attacks, including Return-Oriented Programming and Data-Only attacks. Moreover, our experimental results show that our optimization reduces the overhead by 7% on average compared to a state-of-the-art implementation.

2012 ACM Subject Classification Software and its engineering → Real-time systems software; Security and privacy → Software and application security

Keywords and phrases Real-time system, Software security, Data-flow integrity, Worst-case execution time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.18

Supplementary Material *Software*: `https://gitlab.inria.fr/nbellec1/rt-dfi`

Acknowledgements We want to warmly thank AbsInt for providing the aiT WCET estimator.

1 Introduction

Real-time system (RTS) security has emerged as a growing concern with the increasing development of connected systems such as autonomous vehicles and the Internet-of-things [33, 23]. With many real-time systems still written in memory unsafe languages such as C and C++, the threat of memory corruption bugs remains important [37]. Previous research proved that such vulnerabilities have been used to attack these systems [31, 18].

Protections for RTS have to consider the *Worst-Case Execution Time (WCET)* in their design. In particular, the overhead of the protection on the WCET must be predictable to ensure that the estimation of the WCET remains a safe upper-bound of any execution time.

Previous works on protecting RTS programs against memory corruption have explored the adaptation of *Control-Flow Integrity (CFI)* to real-time constraints [32]. CFI ensures that the protected program execution conforms to the statically computed program's *Control-Flow*

Graph. At each branch executed by the program, CFI verifies that the branch's target belongs to a set of valid targets. The protection considers that any branch to an invalid target corresponds to an attack and responds accordingly (e.g., by stopping the execution). However, CFI cannot protect against advanced attacks that corrupt non-control data. To protect against such threats, we have to rely on protections, such as *Data-Flow Integrity (DFI)* [10, 15], which guards the program against a broader spectrum of attacks.

DFI protects the program by ensuring that every data loaded from memory has been written by a trusted source. Software implementations of this protection are deterministic for a WCET estimator since the whole protection resides in the program's instructions. However, DFI can have a significant time overhead on the protected program, despite the optimizations that have already been proposed [10]. Furthermore, these optimizations have not been designed for RTS and are unaware of the WCET.

In this paper, we present RT-DFI, a new software implementation of DFI that aims at improving the WCET specifically, in contrast to state-of the-art DFI protections. Our approach uses *Integer Linear Programming (ILP)* to reduce the overhead on the estimated *Worst-Case-Execution Path (WCEP)*. Since our objective is to reduce the overhead on the WCET, we focused our work on a single task in the system. We evaluated RT-DFI using aiT [21], the industry standard for static timing analysis. The results show that our approach can reduce the overhead of DFI by up to 18% on the estimated WCET. The contributions of this paper are the following:

- We present RT-DFI, a new method to optimize software DFI for real-time systems. Compared to the state-of-the-art, RT-DFI reduces the DFI overhead on the estimated WCET.
- We implemented RT-DFI and a state-of-the-art DFI protection for a RISC-V processor within the LLVM compilation toolchain.
- We evaluated our method on various tasks of the TACLeBench benchmark suite [17] and showed that we obtained WCET improvements over the state-of-the-art technique, of 7% on average.

The rest of this paper is organized as follows. Section 2 presents some background on real-time systems and DFI. Then, we detail our contribution in Section 3. Section 4 formalizes the constructed ILP used to optimize DFI. Section 5 contains all information on how we conducted our experiments and presents experimental results. Section 6 presents the related work. Finally, Section 7 concludes this paper and proposes some perspectives.

2 Background

Memory corruption attacks aim at modifying the memory of a program to break its security properties, i.e., its confidentiality, integrity, or availability. Nowadays, many attacks have been developed to exploit potential memory corruptions in a program, such as *Return-Oriented Programming (ROP)* [9], *Control-Flow bending* [16], or *Data-Flow bending* [30]. With real-time systems being more connected than ever, malicious actors can exploit vulnerabilities in real-time systems to modify their behavior and break the guarantees of these systems, potentially resulting in significant economic and safety issues.

2.1 Memory Corruption

Since the infamous Morris Worm [3], memory corruption has been a recurring problem. Many countermeasures have been proposed to protect the programs. Some approaches prevent the attacker from accessing crucial information for exploiting program vulnerabilities, such

as Address Space Layout Randomization [35]. In contrast, others aim at detecting and preventing abnormal behavior of the program. Among this second class of protection, many previous works have studied CFI [2, 40], which detects and stops abnormal control-flow of the program.

In practice, CFI detects branches to invalid locations in the program, and it ensures that the return address of a function has not been hijacked. As ROP has become one of the main techniques to exploit memory corruption [9], CFI approaches aim at increasing the difficulty of using this technique. However, Shen et al. has shown that even in the presence of CFI, attackers can hijack the program's data flow to leak confidential data or modify some security-sensitive variables [13]. Castro et al. proposed to enforce the Data-Flow Integrity (DFI) to protect applications from such non-control data attacks [10]. However, the overhead induced by DFI is high. Thus, reducing the cost of DFI for RTS is crucial for the adoption of this protection.

2.2 Data-Flow Integrity

DFI as defined by Castro et al. [10] consists in instrumenting the original program code to assign a unique tag for each store instruction of the program and taint each written byte in memory with the tag of the corresponding store instruction. When the program loads the data later in its execution, DFI checks the data tag to ensure that it belongs to the *valid tag set* of the load. This set contains the tag of all the store instructions that can write the value read by the load. The valid tag set is built at compile-time, using data-flow information. The program is also instrumented to:

- (**A**) Prevent the original application code from accessing the memory area where the tags are stored.
- (**B**) Store the associated tag at each store instruction.
- (**C**) Load and verify the tag at each load instruction, using the *valid tag set*.

Figure 1 presents the steps that the instrumentation code executes at each load and store. We also provide a RISC-V example of an implementation for these steps. When the program executes a load or a store, it first verifies that the target address is neither in the *.text* segment nor in the memory part that contains the tags. The instrumentation code then computes the tag's address to either store or load the tag. If the protected instruction is a store, the instrumented code stores the tag before executing the original store instruction. If the protected instruction is a load, the tag is loaded and then checked against every tag in the *valid tag set* of the load. The initial load is considered legitimate and executed if the tag belongs to the *valid tag set*. Otherwise, DFI detects a data-flow error, considered as an attack.

A naive implementation of DFI can generate a code with more than 100 times the initial execution time [10]. Thus, optimizations have been proposed to reduce the overhead of DFI. The original paper [10] presents three optimizations: **equivalent classes optimization**, **redundant elimination**, and a greedy **tag check optimization**.

The **equivalent classes optimization** reduces the overall number of tags by detecting tags that always appear together in the same *valid tag sets* and regrouping them as a unique tag. This optimization reduces the overall number of tags used and, more importantly, reduces the number of checks required when a load occurs. In the original paper by Castro et al. [10], this optimization is the most important one, reducing the overhead of DFI below ten times the original execution time.

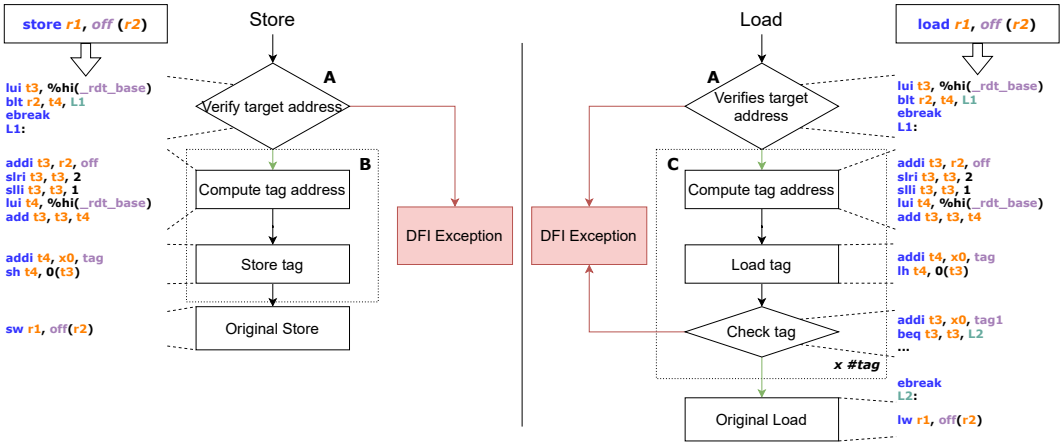

Figure 1 DFI protection instrumentation. The left part represents the instrumentation of store instruction. The right part represents the instrumentation of load instructions. For each, we give an example of RISC-V implementation.

The **redundant check elimination** statically analyses the original program code to find successive loads or stores in the same basic block which target the same memory cell. When it discovers such successive instructions, it removes redundant protections. In the case of two successive loads, it removes the protection on the second load if the *valid tag set* of the first load is a subset of the *valid tag set* second. In the case of two consecutive stores with the same tag, we can remove the instrumentation for the second store since it would just overwrite the tag with itself. This optimization removes parts or the complete protection on loads and stores when it can statically ensure that the protection is redundant.

The **tag check optimization** improves the checking of tags for each load by checking multiple tags simultaneously. For this optimization, we differentiate between the tag (t), an identifier of the store that writes into a memory location, and the **tag representation** (r_t), an unsigned number encoding the tag, which the optimization can modify. Rather than verifying each tag of *valid tag set* one after the other in a *valid tag set*, we can regroup consecutive tag representations to form intervals. In this case, we can compare the tag of the loaded data with the limits of intervals defining the *valid tag set*. Checking intervals is particularly efficient for the biggest valid tag sets. Thus, the tag check optimization modifies the tag representations to reduce the number of intervals to check.

In the original article by Castro et al. [10], this last optimization uses a greedy algorithm that sorts the *valid tag sets* by decreasing value of the following metrics: $use \times size$ with use the number of load instructions that use the *valid tag set* and $size$ the size of the *valid tag set*. The algorithm then iterates on all the *valid tag sets*, in the order given by the metric, and assigns consecutive tag representations to the tags in each *valid tag set*, skipping the tags that already received a tag representation in a previous *valid tag set*.

The first issue of this algorithm is that it does not use any information about execution paths. In particular, loop bounds or branch frequencies could help to refine the representation allocation to focus on sets that are checked more frequently and not just that appear on more distinct loads. The second issue is that all the tags in a given set are assigned a new representation (except the tags that already have a new one) before analyzing another set. This local optimization prevents improvement based on multiple sets. For example, given the sets $\{A, B, C\}$ and $\{A, D, E\}$, the optimization may treat the first set and assign 1, 2, 3 to

respectively A, B, C and then consider the second set and assign 4 and 5 to respectively D and E. In this case, it would be more interesting to assign 3 to A such that the second *valid tag set* can also correspond to an interval.

Table 1 Valid tag sets optimizations.

Valid tag sets	Tag representations
$S_1 = \{A, B, C, E\}$ $S_2 = \{A, B, E\}$ $S_3 = \{A, B, D\}$	$A \to 1$ $D \to 4$ $B \to 2$ $E \to 5$ $C \to 3$
Equivalent classes A∼B	
$S_1 = \{A, C, E\}$ $S_2 = \{A, E\}$ $S_3 = \{A, D\}$	$A \to 1$ $D \to 4$ $C \to 3$ $E \to 5$
Tag check optimization	
$S_1 = \{A, C, E\} \sim [\![1, 3]\!]$ $S_2 = \{A, E\} \sim [\![1, 1]\!] \cup [\![3, 3]\!]$ $S_3 = \{A, D\} \sim [\![1, 1]\!] \cup [\![4, 4]\!]$	$A \to 1$ $D \to 4$ $C \to 2$ $E \to 3$
Optimal tag check	
$S_1 = \{A, C, E\} \sim [\![1, 3]\!]$ $S_2 = \{A, E\} \sim [\![2, 3]\!]$ $S_3 = \{A, D\} \sim [\![3, 4]\!]$	$A \to 3$ $D \to 4$ $C \to 1$ $E \to 2$

Table 1 presents the effect of the optimizations on *valid tag sets* on a simple example. The first column describes the *valid tag sets*, and the second column gives the tag representations. Dashed lines separate each optimization. We start with an example of three sets S_1, S_2, and S_3. The *equivalent classes optimization* finds that tags A and B are equivalent and regroups them into tag A without modifying the tag representations, which creates a gap between the tag representations of A and C in S_1. Then, the *tag check optimization* modifies the tag representations to remove this gap but, as it operates set by set, it fails to notice that A belongs both to S_1 and S_3. Thus, the optimization cannot close the gap between the tag representation of A and D when it passes on S_3. Finally, the figure presents a more sophisticated optimization that we would like to obtain.

Our work uses an ILP solver combined with information on the WCEP to tackle the two issues of *tag check optimization*. Notice that our goal is to improve the WCET rather than the average runtime.

2.3 Real-Time systems

RTS are computer systems with timing constraints, often due to their interaction with physical components. To ensure the respect of these constraints, we rely on analysis that estimates the WCET of each task. The WCET is then used with other scheduling data to verify that every task in the system has enough time to complete its execution. WCET estimation is performed on the executable file of the task to have all the low-level data available (e.g., processor instructions executed and the address of these instructions) and for a specific architecture to consider micro-architectural effects. As commonly accepted in the RTS community, WCET estimation is performed *in isolation*, leaving estimation and consideration of interferences between tasks to the schedulability analysis step.

Multiple analyses results are combined to perform WCET estimation. In particular, symbolic analyses of the cache, memory, and registers improve the precision of the WCET analysis. The value analysis safely determines the targeted memory area(s) of the instruction for each load and store of the program and the value that this instruction writes/reads in this memory area. This analysis results depend on the **context** of the instruction (e.g., the state of the call stack or the number of iterations of the loop containing the instruction). The analyses may over-approximate the results to avoid consuming too many resources (memory and time) and always provide a sound result. For instance, the value analysis provides a superset of actual memory locations. The returned set may also contain values that do not appear at run-time. The WCET is estimated using the results of these analyses for each basic block in the program. A path called WCEP, which maximizes the sum of the time of the basic blocks, is computed with a technique based on ILP called *Implicit Path Execution Technique (IPET)*.

We can retrieve much data from the WCET estimation as a byproduct of these analyses. In particular, knowing the WCEP allows targeting optimizations along this path, thus reducing the WCET. Such WCEP-oriented optimizations may result in a change of the WCEP, motivating iterative solutions to tackle WCEP-changes. RT-DFI uses path information to iteratively reduce the cost of the tag check of DFI .

3 Overview of RT-DFI, a WCET-directed DFI scheme

Decreasing the WCET of tasks improves the schedulability of a task set. Thus, we give in this section an overview of our approach named RT-DFI, which reduces the WCET of individual DFI-protected tasks. We designed RT-DFI such that it has the same level of protection as the original DFI. RT-DFI reduces the cost of tag check verifications using WCEP-oriented optimization. The optimization is iterative to tackle WCEP changes.

3.1 Using WCEP information to optimize tag checks

We focus our work on reducing the overhead of the tag check part of DFI, as this is the primary source of overhead [20]. The tag check overhead can be decomposed into two factors. The first factor is the number of checks required to cover the whole valid tag set of the load. Since the tags are checked using intervals, we can improve this factor by modifying the representation of the tags to reduce the number of intervals that cover the *valid tag set*. The second factor is the order in which we check these intervals. The tag check verifies if the tag belongs to each interval covering the *valid tag set*, and jumps to the rest of the code as soon as it finds an interval containing the tag. Thus, we can reduce the tag check overhead by first checking the most frequent intervals. In the rest of this paper, we use *interval order* to describe the order of the checks against the intervals.

RT-DFI uses the context data (number of executions in the WCEP, value analysis results) to optimize these two factors. To improve the WCET, we perform the optimization specifically on the WCEP. Thus, we only focus on contexts present in the WCEP. This approach also decreases the number of loads/contexts that we consider, reducing the complexity of our optimization problem.

In the rest of this paper, we assume that the WCET solver performs a symbolic value analysis and that we have access to its results. The WCET solver uses this value analysis to improve the IPET, avoiding paths in each context where the value analysis detects that these paths are infeasible. For example, the value analysis may establish that a condition is always satisfied in some context C_0. In this case, the IPET only considers the path when the

condition holds for the context C_0, even if the alternative path is more costly for the IPET in general. The same principle applies to the tag checks, which are a series of conditions. Thus, although all the tag checks are present in the program code, the value analysis may refine the IPET to bypass some checks in some contexts. When the value analysis cannot restrict the possible tag values for a given load, it still considers that the possible values are restricted by the **valid tag set** of the load since it reasons only on the legitimate executions of the program.

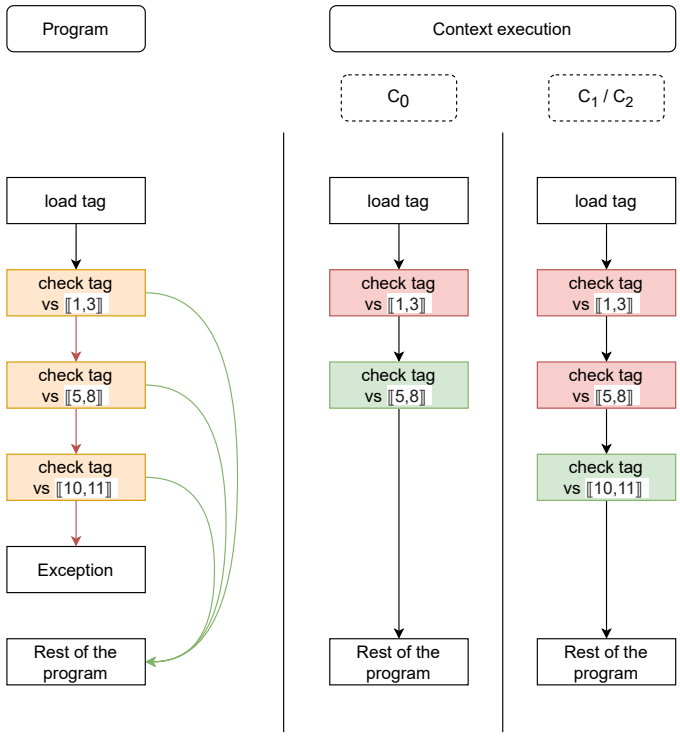

Figure 2 Example of how checks can be bypassed in the WCEP.

For example, we present in Figure 2 a tag check with three intervals, $[\![1,3]\!]$, $[\![5,8]\!]$, and $[\![10,11]\!]$. We also present the path considered by the IPET for three contexts C_0, C_1, and C_2 without optimization. We want to show how we can use the information from the value analysis to improve DFI overhead. Suppose the value analysis infers that the loaded tag is equal to 7 in context C_0, is either 1 or 10 in context C_1, and can take any value in context C_2. For C_0, the IPET considers the path that performs the first two checks and then jumps to the rest of the code, since the value analyses inferred that the tag belongs to the interval of the second check. Thus, the check of $[\![1,3]\!]$ and $[\![5,8]\!]$ count for the time of this context but not the check of $[\![10,11]\!]$. In this case, it would be more optimal to place the check of $[\![5,8]\!]$ as the first check for this context, as the IPET would skip the two other checks for the context C_0. For C_1, the IPET has to take into account the two possible values of the tag and can only jump to the rest of the code after the check of $[\![10,11]\!]$. In this context, one way to optimize the WCET is to modify the interval orders to place $[\![1,3]\!]$ and $[\![10,11]\!]$ as the two first intervals (whatever the order of these two) to skip the check of the last interval. Another optimization is to modify the tag representations such as the new representations of 1 and 10 belong to the same interval. In this case, we can place this new interval as the first one to check, effectively skipping all other intervals. For the last context C_2, the IPET

cannot skip any check, as it has no information on the tag value (except that it should be inside the *valid tag set*). Thus, the only way to optimize C_2 is to reduce the number of intervals that cover the *valid tag set*, by modifying the tag representations.

This example shows that different optimizations are possible in function of the information provided by the value analysis. We can change the interval orders, we can modify the tag representations, or do both. Modifying the interval orders only has a local impact on the WCEP. On the other hand, modifying the tag representations has consequences not only on the load we are focusing on, but also on every other load that has any of these tags in its *valid tag set*. Thus, knowing if a tag representation modification is interesting is more complex than for an interval order modification. Furthermore, multiple contexts for the same load can have conflicting optimizations. In our example, contexts C_0 and C_1 can both be optimized by modifying the interval orders, but the optimal order of C_0 conflicts with the optimal order of C_1. To deal with this problem, we use an ILP solver that provides a good solution along the whole WCEP, taking into account global effects when modifying the tag representations.

To deal with the three possibilities seen in the previous example, we divide the contexts of the value analysis into three types, depending on the result of the value analysis:

(a) The value analysis has determined exactly a single tag for this context (**known tag context**). In this case, we know that only one test will be true and once this test is executed, we jump to the rest of the program. Thus, we want to execute this test as soon as possible. This case corresponds to context C_0 in our previous example.

(b) The value analysis found multiple possible tags, which are a strict subset of the *valid tag set* (**partially known tag context**). In this case, checks are skipped only when all the tags of the context are tested. Thus, we are interested in placing the tests for these tags as soon as possible, but the order between these tests is of no importance (for this context). Modifying the tag representations to reduce the number of intervals is also interesting. This case corresponds to context C_1 in our previous example.

(c) The value analysis was unable to refine the possible values (i.e., the whole *valid tag set* is possible) (**unknown tag context**). In this case, no test can be skipped. Thus, the only way to reduce the cost of such context is to reduce the number of tests by finding a better tag representation for this *valid tag set*. This case corresponds to context C_2 in our previous example.

These different types of context represent how much information we have about if and when tests are skipped. The goal is then to use the context data to improve the tag representation and the interval orders on the WCEP. To do so, we use an ILP model that searches for a minimal number of tests along the WCEP. In contrast with the greedy algorithm of the first *tag check optimization* described in section 2.2, an ILP can provide better solutions that use all the information we have (see Section 5). In particular, while the greedy algorithm only performs local improvement load per load, an ILP can find optimizations that have a global impact on multiple loads. Furthermore, the current ILP solvers are very efficient and propose to set a timeout to stop the resolution after a given time, which can be used to obtain a good solution while having a bound on the time consumed by the solving algorithm.

The optimization flow is the following: the program is compiled the first time with DFI and all the optimizations present in the original paper by Castro et al. [10]. We then use the WCET analysis to construct an ILP that we optimize to find a better solution on the WCEP. We modify the program with this new solution. We then repeat the optimization process by creating a new ILP based on the new program executable, while still maintaining the same level of optimization on the previous paths. This allows RT-DFI to converge as the WCET reduces or stays the same. The process stops when the WCET does not improve anymore or after a given number of iterations.

3.2 Principle of the ILP

To optimize the WCEP, we use an ILP solver to minimize the number of checks on the current WCEP. We use the ILP to optimize the tag representation and, for each load on the WCEP, to find an optimal interval order. To do so, we use the result of the value analysis for each load in the WCEP. In particular, for each load, we segregate the contexts into known/partially known/unknown tag contexts, and we retrieve the **weight** of each context in the WCET, i.e., the number of times each context is executed in the WCEP. We also assume that check costs are the same on the WCET, whether they check against an interval or a single tag. This allows us to model single tags as singleton intervals in our ILP. This assumption is true in our implementation (see Section 5), where we generate code that checks intervals as fast as a single tag using only one branch instruction (with a method explained in [10]).

We construct the ILP with three steps:

1. We find a valid tag representation for each tag in the WCEP. We represent each tag as a variable, and we add constraints to prevent two tags from having the same value.
2. We find an optimized interval order for each load in the WCEP. For each load, we generate one variable per tag in the *valid tag set* of this load instruction, this variable representing the interval order of the interval containing the tag. We add constraints to these variables such that tags that must be in the same interval have the same order and tags that are in different intervals have different orders.
3. We aggregate the order variables of each load into a weighted sum that represents the objective function to optimize. For every context of a given load, the number of checks is the maximum of the order variables of the possible tags of the context (i.e., how much interval must be checked before we can skip the rest) multiplied by the **weight** of the context.

For example, in Table 2, we present four tags with their tag representations (for the entire program) and their interval orders (for a given load l). We see that for this load, tags A and B are tested first, then tag C and finally tag D. We also present 3 contexts C_0, C_1 and C_2 with their possible tags and the cost associated with each context as well as the overall cost. The context C_0 has B as its only possible tag, and the WCEP only checks against the first interval before jumping to the rest of the code. Thus, the cost of C_0 is 1 (the interval order of B) times w_0 (the weight of C_0). For context C_1, which has more possible tags (C and D), the WCEP passes by the three checks before jumping to the rest of the code. Thus, the cost of this context is 3 times w_1. Finally, for an unknown context such as C_2, the WCEP is forced to check against all the intervals. Thus, it has a cost of 3 times w_2. The objective function is the sum of each context cost for each load.

To deal with changes of the WCEP we add new constraints to our ILP that prevent new optimizations from destructing the previous ones, which allows RT-DFI to converge. These new constraints have the following shape: '*previous objective function*' $\leq$ '*previous objective function value*'. The '*previous objective function*' is constructed the same way as the current objective function, but with the context of the previous optimization. The '*previous objective function value*' is just the minimal value of the objective function found by the previous optimization. These constraints force the ILP to optimize the current WCEP while maintaining the same level of optimization on the previous path. Of course, if we have multiple previous WCEPs, we can add one constraint of this shape per previous WCEP.

Table 2 An example of how the cost of the contexts are computed by the ILP for an arbitrary load l, knowing a tag representation and the interval order.

(a) Tag representations and interval order.

Tag	Representation	Interval Order (for load l)
A	2	1
B	3	1
C	5	2
D	9	3

(b) Contexts and associated costs. w_0, w_1 and w_2 represent the *weight* of the contexts (the number of time they appear in the WCEP).

Context	Tags	Cost
C_0	B	$\max(1) \cdot w_0$
C_1	C,D	$\max(2,3) \cdot w_1$
C_2	A,B,C,D	$\max(1,2,3) \cdot w_2$
All	A,B,C,D	$\max(1) \cdot w_0 +$ $\max(2,3) \cdot w_1 +$ $\max(1,2,3) \cdot w_2$

4 Formal definition of the ILP for WCET-oriented tag check optimization

In this section, we present a formal definition of the ILP we use to optimize DFI on the WCEP. We first give a notation table and describe which data is available to construct the ILP in Subsection 4.1. We explain in Subsection 4.2 how to compute the number of checks for a load when the tag representations and the order of the intervals are known. We then present in Subsection 4.3 the ILP that minimizes the number of checks on the WCEP by optimizing the tag representations and the order of the intervals at the first iteration of the algorithm. Finally, we explain in Subsection 4.4 the constraints we add to the ILP to handle potential WCEP changes.

4.1 Notation table and problem formal definition

Table 3 Notation table for the mathematical terms.

Notation	Type	Signification
$[\![a,b]\!]$	Interval	**Integer** interval between a and b
l	Load	A protected load
L	Set[Load]	Set of loads in the WCEP
t	Tag	A tag
T	Set[Tag]	Set of tags checked in the WCEP
r_t	$\mathbb{N}$	The tag representation of t (fixed)
s_l	Set[Tag]	Valid tag set of load l
$I_{l,t}$	Interval	Interval specific to l containing r_t
$\phi_{l,t}$	$\mathbb{N}$	Order of $I_{l,t}$ (check order) (fixed)
C_l	Context	A context for l in the WCEP
T_{C_l}	Set[Tag]	Possible tags for the context C_l
w_{C_l}	$\mathbb{N}$	Number of occurrence of C_l in the WCEP
N_l	$\mathbb{N}$	Number of checks of l in the WCEP
$Succ(t)$	Tag	t' such as $r_{t'} = r_t + 1$

Two tables are used to formally describe the ILP. Table 3 contains the mathematical terms we use. Note that when we use r_t or $\phi_{l,t}$, we consider them already fixed. Table 4 lists the ILP variables and constants used in the ILP formulation. A few notations represent values of the mathematical domain. The difference with the ones listed in Table 3 is that they are determined by the ILP solver and not fixed. We note them as *free*.

Table 4 Notation table for ILP variables and constants.

Notation	Type	Signification
M	Constant	Number greater than any factor in the ILP (see big-M notation [22])
$start$	Constant Tag	Special tag used as the start of tag representation
end	Constant Tag	Special virtual tag used as the end of tag representation
V	Set[Tag]	$T \cup \{start, end\}$
v_t	$\mathbb{N}$	Vertex representing tag t
$entry_t$	$\mathbb{N}$	Number of edges entering v_t
$exit_t$	$\mathbb{N}$	Number of edges exiting v_t
$e_{t,t'}$	$\mathbb{B}$ (boolean)	There is a directed edge from t to t' (or not)
R_t	$\mathbb{N}$	Represent r_t (free)
$\lambda^+_{l,t}$	$\mathbb{B}$	Represents if $Succ(t) \in s_l$ or not
$\Lambda^+_{l,t,t'}$	$\mathbb{B}$	$\lambda^+_{l,t}$ if $R_t < R_{t'}$ else $\lambda^+_{l,t'}$
$\Phi_{l,t}$	$\mathbb{N}$	Represent $\phi_{l,t}$ (free)
$\Phi^+_{l,t}$	$\mathbb{N}$	$\Phi_{l,t'}$ for t' such as t' = $Succ(t)$
$\Delta_{l,t,t'}$	$\mathbb{N}$	Represents $\|\Phi_{l,t} - \Phi_{l,t'}\|$
$\Delta^+_{l,t,t'}$	$\mathbb{N}$	Represents $\|\Phi^+_{l,t} - \Phi_{l,t'}\|$
$\Gamma_{l,t,t'}$	$\mathbb{N}$	$\Delta^+_{l,\alpha,\beta}$ if $Succ(\alpha) \in s_l$ else 0 with $\alpha, \beta \in \{t, t'\}$, $R_\alpha < R_\beta$

We first recall the problem at hand and we explicit which data we have before the ILP. We then dive into the formal construction of the ILP in the next subsections. Our goal is to construct an ILP that can select the tag representations (globally) and interval orders (for each load on the WCEP) to minimize the number of checks on the current WCEP. To do so, we have data on all the contexts of each load in the WCEP. For each context C_l of the load l, present on the WCEP, we have the result of the value analysis (in the worst-case, the result is the valid tag set s_l of l) as well as the number of occurrences of C_l in the WCEP. When we want to iterate the optimization after a change of WCEP, we also consider that we have the same kind of data for the previous WCEP (as we just optimized it) as well as the value of the objective function of the last iteration. These data are used in subsection 4.4 to never undo previous optimizations.

4.2 Computing the number of checks

For this part, we consider that we know every tag representation, written r_t. We regroup the tag representations into intervals for each load l and we assign an arbitrary order to these intervals. Note that we can map s_l to a minimal number of intervals containing only valid tags. To do so, we consider all tags are single-element intervals and we merge adjacent intervals until there is no more fusion possible. We note $I_{l,t}$ the interval specific to l that contains r_t. As the intervals can be checked in an arbitrary order, we assign to each interval $I_{l,t}$ an index $\phi_{l,t}$ (starting at 1) which represents the order of the checks of the intervals (the interval with index 1 is checked first then the one with index 2, etc.). Note that for two tags t and t', if $r_t \in I_{l,t'}$ then $I_{l,t} = I_{l,t'}$ and $\phi_{l,t} = \phi_{l,t'}$.

▶ **Example 1.**

t	r_t	$I_{l,t}$	$\phi_{l,t}$
A	1	$[\![1,2]\!]$	1
B	2	$[\![1,2]\!]$	1
C	4	$[\![4,4]\!]$	2

We provide an example with 3 tags A, B and C in Example 1. As the tag representations are assigned for the entire program, the tag representations of A, B and C have no reason to be contiguous. We provide the interval of each tag and an arbitrary index for each interval.

Let C_l be a context for the load l with T_{C_l} the set of values provided by the value analysis and w_{C_l} the number of occurrences of C_l in the WCEP. We obtain the following number of checks for C_l in the WCEP:

$$\max_{t \in T_{C_l}} (\phi_{l,t}) \cdot w_{C_l} \tag{1}$$

This formula appears because the IPET only bypasses checks once they have covered all the possible tags of the context, and because the context appears w_{C_l} times in the WCEP.

The number of checks for a given load is an aggregation of the number of checks for every context of this load in the WCEP (2). The number of checks over the whole WCEP is the sum over all the loads (3).

$$N_l = \sum_{C_l} (\max_{t \in T_{C_l}} (\phi_{l,t}) \cdot w_{C_l}) \tag{2}$$

$$\sum_{l \in L} N_l \tag{3}$$

4.3 Transformation into an ILP problem

In the previous part, we described how to compute the number of checks once the tag representations and the interval orders are known. Thus, to construct the ILP, we only need to construct these two components. As we describe the ILP, we use a few notation shortcuts, $(x < y)$, $(x \cdot b)$ and $max(x, y, ...)$ to represent variables with the same value as these functions. For more information on how to construct such variables, we refer to [22].

We choose the tag representations to form a contiguous interval since this mapping requires the lesser space to store the tags during the program execution, and it maintains the space overhead of DFI.

In this case, we can sort the tags by their representation. Moreover, every tag, except the first and last tags of the interval, has a successor and a predecessor. These are precisely the properties of a path in a graph, which is easily expressible as an ILP. Furthermore, since, in principle, every tag can have any representation, we need a complete graph to allow all paths. Thus, we use a complete directed graph to construct the tag representations, whose vertices v_t represent the tags t.

In this graph, we want to select a vertex-cover path, which provides us with the tag representations. For a given path, the edge $e_{t,t'}$ from v_t to $v_{t'}$ is present if $r_{t'} = r_t + 1$. To ease the construction of the ILP, we introduce two virtual tags, *start* and *end*, which are the start and end of the path.

In Figure 3, we present an example of the graph for 4 tags, plus *start* and *end*. We also give an example of a path in this graph from v_{start} to v_{end}, and the corresponding tag representation mapping. We obtain this mapping by following the path and assigning consecutive representations to each tag.

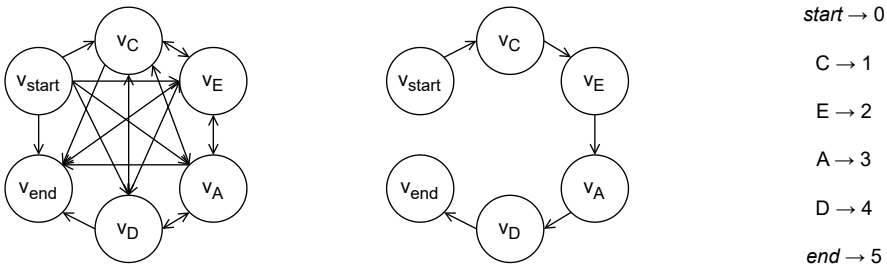

Figure 3 Example of the tag representation by the ILP.

We introduce $entry_t$ (resp. $exit_t$), which counts the number of edges entering (resp. exiting) the vertex v_t as well as R_t the variable containing the tag representation of t. The constraints for the path are the following[1][2]:

$$\sum_{t,t' \in V} e_{t,t'} = Card(V) - 1 \quad (4)$$

$$\forall t \in V, entry_t = \sum_{t' \in T \setminus \{t\}} e_{t',t} \quad (5)$$

$$\forall t \in V, exit_t = \sum_{t' \in T \setminus \{t\}} e_{t,t'} \quad (6)$$

$$\forall t \in T, entry_t = 1 \quad (7)$$

$$\forall t \in T, exit_t = 1 \quad (8)$$

$$\forall t, t' \in V, (R_{t'} + 1) - Card(T) \cdot (1 - e_{t',t}) \leq R_t \leq (R_{t'} + 1) + Card(T) \cdot (1 - e_{t',t}) \quad (9)$$

$$entry_{start} = 0, exit_{end} = 0, R_{start} = 0, R_{end} = Card(V) - 1 \quad (10)$$

Constraint (4) forces the path to have no more edges than necessary for a path passing by each vertex only once. Constraint (5) (resp. (6)) defines $entry_t$ (resp. $exit_t$). Constraint (7) (resp. (8)) forces each vertex except v_{start} (resp. v_{end}) to have only 1 entry edge (resp. 1 exit edge). Constraint (9) forces $R_t = R_{t'} + 1$ if and only if v_t is the vertex next to $v_{t'}$ in the path. Finally, constraint (10) deals with the special cases of tags *start* and *end*. The overall design of this part of the ILP is a classic directed graph representation [14] combined with constraints (9) and (10).

We now explain how the ILP computes the interval orders. We create for each tag t and each load l a variable $\Phi_{l,t}$ that contains the interval order of $I_{l,t}$. In the case t does not belong to s_l, we assign an arbitrary number to $\Phi_{l,t}$ such that two different tags (both not belonging to s_l) have different indexes and such that these indexes are higher than the maximum interval index of l (i.e., greater than $Card(s_l)$). As the ILP computes the tag representations R_t, it must also compute the intervals and their orders, as the number of intervals and the interval themselves depends on the tag representations. We implicitly define the intervals with the following lemma:

▶ **Lemma 2.** $\forall t, t' \in T, I_{l,t} = I_{l,t'} \iff \phi_{l,t} = \phi_{l,t'}$

Lemma 2 expresses that if two tags are in the same interval, then they have the same index. Thus, we can encode in the ILP that two tags t and t' are in the same interval for the load l if and only if $\Phi_{l,t} = \Phi_{l,t'}$.

[1] We use the classic encoding of boolean variable in ILP with $false = 0$ and $true = 1$
[2] $Card(S)$ the cardinal of S

Writing constraints that represent if two tags t and t' are in the same interval (for a given l) is complex in the ILP, as it requires checking that every tag with a representation in between R_t and $R_{t'}$ belongs to s_l. To handle this problem, we use Lemma 3.

▶ **Lemma 3.** $\forall a \leq b, \forall S \subset \mathbb{N}, [\![a,b]\!] \subset S \iff (a \in S) \wedge ([\![a+1,b]\!] \subset S)$

Rather than verifying that all the tags in between R_t and $R_{t'}$ belongs to s_l, we just verify that the $Succ(t)$ (considering that $R_t < R_{t'}$) belongs to s_l and let another part of the ILP handle the verification of a smaller interval. We can then recursively use the same Lemma until we have no more tags in between R_t and $R_{t'}$ to check.

We can thus rewrite these two lemmas to obtain the following relation:

$$\forall t, t' \in s_l, t \neq t', r_t < r_{t'}, \phi_{l,t} = \phi_{l,t'} \iff \phi_{l,Succ(t)} = \phi_{l,t'} \tag{11}$$

with $Succ(t)$ the tag t' such that $r_{t'} = r_t + 1$. Relation 11 explains that we can infer that two tags belong to the same interval by knowing if the successor of one of the tags belongs to this interval, as long as a few conditions are met. As we do not know whether $R_t < R_{t'}$ before executing the ILP, we use ILP variables that represent $\|R_{l,t} - R_{l,t'}\|$[3] and we build constraints that enforce Relation 11.

$$\forall t \in s_l, \Phi^+_{l,t} = \sum_{t' \in T} (e_{t,t'} \cdot \Phi_{l,t'}) \tag{12}$$

$$\forall t \in V, \lambda^+_{l,t} = \sum_{t' \in s_l} e_{t,t'} \tag{13}$$

$$\forall t, t' \in s_l, \Lambda^+_{l,t,t'} = (R_t < R_{t'}) \cdot \lambda^+_{l,t} + (R_{t'} < R_t) \cdot \lambda^+_{l,t'} \tag{14}$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} = (\Phi_{l,t'} < \Phi_{l,t}) \cdot (\Phi_{l,t} - \Phi_{l,t'}) + (\Phi_{l,t} < \Phi_{l,t'}) \cdot (\Phi_{l,t'} - \Phi_{l,t}) \tag{15}$$

$$\forall t, t' \in s_l, \Delta^+_{l,t,t'} = (\Phi_{l,t'} < \Phi^+_{l,t}) \cdot (\Phi^+_{l,t} - \Phi_{l,t'}) + (\Phi^+_{l,t} < \Phi_{l,t'}) \cdot (\Phi_{l,t'} - \Phi^+_{l,t}) \tag{16}$$

$$\forall t, t' \in s_l, \Gamma_{l,t,t'} = (R_t < R_{t'}) \cdot \lambda^+_{l,t} \cdot \Delta^+_{l,t,t'} + (R_{t'} < R_t) \cdot \lambda^+_{l,t'} \cdot \Delta^+_{l,t',t} \tag{17}$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} <= \Gamma_{l,t,t'} + (1 - \Lambda^+_{l,t,t'}) \cdot M \tag{18}$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} >= \Gamma_{l,t,t'} + (1 - \Lambda^+_{l,t,t'}) \tag{19}$$

Constraint (12) defines $\Phi^+_{l,t}$, a variable that represents $\phi_{l,Succ(t)}$. Constraint (13) defines $\lambda^+_{l,t}$, a binary variable equal to 1 if and only if $Succ(t) \in s_l$. Constraint (14) defines $\Lambda^+_{l,t,t'}$, a binary variable equals to $\lambda^+_{l,a}$ with a the tag with the lowest tag representation between t and t'. Constraint (15) (resp. (16)) defines $\Delta_{l,t,t'}$ (resp. $\Delta^+_{l,t,t'}$) which represents $\|\phi_{l,t} - \phi_{l,t'}\|$ (resp. $\|\phi_{l,Succ(t)} - \phi_{l,t'}\|$). Constraint (17) defines $\Gamma_{l,t,t'}$, which represents $\Delta^+_{l,\alpha,\beta}$ if $Succ(\alpha) \in s_l$ else 0, with $\alpha \neq \beta \in \{t, t'\}$ s.a. $R_\alpha < R_\beta$. Finally, constraint (18) (resp. (19)) provide an upper bound (resp. lower bound) on $\Delta_{l,t,t'}$ that enforces the relation expressed in (11) and handles the case where $Succ(t)$ and/or $Succ(t')$ are not in s_l.

All these constraints organize the $\Phi_{l,t}$ variables such that we obtain the intervals and their orders. We can then use the $\Phi_{l,t}$ variables to build the objective function (3) (seen in Subsection 4.2) that we want to minimize.

4.4 Handling WCEP changes

When optimizing for the WCET, we must handle changes of the WCEP. In particular, incremental optimizations must ensure that an iteration does not destroy the work of a previous iteration. To do so with our ILP, we use additional constraints that prevent

[3] $\|x\|$ being the absolute value of x

increasing the number of checks on the previous WCEPs. These constraints are of the form $objective_i <= result_i$ with $objective_i$ being the objective function of the previous i^{th} iteration and $result_i$ the minimal value of this objective function, found by the previous iteration of the ILP solver.

As we use the same kind of variable as the real objective function, all the constructions explained previously remain the same. This approach also allows us to reduce the complexity of the ILP when the same tags/loads appear in two distinct WCEP (be it the previous or current one) as we can use the same variables and constraints to represent both.

5 Experimental results

To validate RT-DFI, we implemented it for a RISC-V LLVM compiler toolchain and applied it to a series of benchmarks from the TACLeBench benchmark suite. In subsection 5.1, we present how we implemented RT-DFI[4] and the baseline DFI protection it is compared against, the benchmarks we used and the methodology. In subsection 5.2, we provide and analyze the results of this experiment. We also comment in subsection 5.4 on some DFI violations that we encountered in the benchmarks.

5.1 Experimental setup

We implemented RT-DFI using **LLVM** [28] as a compilation chain. **PhASAR** [34] produces the *valid tag set*. The compilation process is performed in the following way:

1. We transform the source code into *LLVM Intermediate Representation (IR)* with **O1** optimization switch (to avoid unnecessary load/store instructions that greatly increase the overhead of DFI).
2. We use *PhASAR* to produce the *valid tag sets* used by DFI and integrate them into the *LLVM IR* as annotations.
3. We apply the optimizations described in Section 2 on the *LLVM IR* to obtain the state-of-the-art DFI presented in [10].
4. We compile the *IR* into a *RISC-V* 32-bits executable for the *RudolV*[5] processor. We integrated a last pass to the *RISC-V* backend of *LLVM* that transforms DFI annotations into assembly code. This prevents any separation between the protection and the protected instruction and handles the case where the instruction was not present in the *IR* (register scavenging, entry/exit of functions).
5. We execute RT-DFI, which produces a new IR that we can feed to the previous step.

This required to change $\sim 5,000$ lines of code in *LLVM* and $\sim 1,000$ lines of code in *PhASAR*. We use the industry standard for static timing analysis **aiT** [21] to compute the WCET, obtain the WCEP and perform value analysis. *CPLEX* [5] 20.1 is used to solve the ILP problems described in section 4. The rest of the implementation is written in Python 3.8. It orchestrates the programs (aiT, CPLEX, LLVM, PhASAR), it aggregates and unifies data retrieved from aiT and the compilation chain (in particular, the *valid tag sets*), it generates the ILP and finally, it modifies the *LLVM IR* to handle the new optimized tag representations and interval orders. This represents $\sim 8,000$ lines of code in Python.

For our experiments, we use TACLeBench [17]. This benchmark suite is composed of five groups of benchmarks:

[4] `https://gitlab.inria.fr/nbellec1/rt-dfi`
[5] `https://github.com/bobbl/rudolv`

1. **kernel** contains small kernel functions such as a binary search or an md5 hash.
2. **sequential** contains large function blocks (e.g. cryptographic or compression algorithms).
3. **test** contains three synthetic programs designed to challenge the WCET analysis tools.
4. **parallel** contains two modified real-world parallel applications, *Debie* and *PapaBench*.
5. **app** contains two real applications, *lift* and *powerwindow*.

A full benchmark description can be found in [17]. Out of the five groups, we did not run our experiments on the **parallel** group as we consider a bare-metal execution of the programs without relying on a Real-Time Operating System (RTOS), which defeats the purpose of the **parallel** group. We also excluded the *bitonic*, *bitcount*, *fac*, *quicksort*, *recursion*, *ammunition*, *anagram* and *huff_enc* benchmarks as it was harder to obtain the recursion bounds for the aiT WCET solver. Furthermore, *rijndael_dec* and *rijndael_enc* fail to pass *aiT* as they violate DFI, as explained in subsection 5.4.

We focus our experiment on the **aiT estimated WCET** after executing RT-DFI compared to the state-of-the-art DFI implementation. We perform two optimizations based on the data provided by aiT. The first optimization uses the value analysis of aiT to improve the *valid tag set* of our protection. In some cases, we can further reduce the *valid tag sets* provided by PhASAR by using the value analysis of aiT. In particular, the data-flow analysis used with PhASAR is field-insensitive (as [10]), and thus fails to distinguish between the cells of an array, which forces the analysis to keep some tags that could be removed. On the other hand, the value analysis of aiT is performed at the memory level and can help to refine the *valid tag set* of PhASAR. However, we can not use only aiT to obtain the *valid tag sets*, as their construction use information only available in the LLVM IR. This improvement of the *valid tag sets* is a byproduct of using the value analysis of a WCET solver to optimize the WCEP. We only present it in this section as the same result could be obtained by improving our data-flow analysis. Since it provides interesting results, we ought to present it. The second optimization is RT-DFI, presented in Section 4. For this second optimization, we executed four iterations for each benchmark, to address potential WCEP changes. However, we did not see improvement past the first iteration, so we just provided the improvement after the first iteration in our analysis of the results. We did not bound the ILP runtime, as all ILP were solved in less than 40 seconds.

5.2 Results

Figure 4 shows the normalized overhead factor of the state-of-the-art DFI on the WCET, with 1 the WCET of the program without DFI. 20 out of the 47 benchmarks have an overhead between $\times 1.03$ and $\times 1.6$ while the remaining 27 benchmarks have an overhead between $\times 1.9$ and $\times 5$. The mean overhead is $\times 2.38$. As observed in [10], DFI can incur a high overhead on the protected program, depending on the number of store/load instructions of the program. This confirms the need to reduce the impact of DFI on the WCET.

Figure 5 shows for each benchmark the improvement on the WCEP in percentage compared to the state-of-the-art when improving *valid tag sets* with the value analysis (*value analysis improvement* in the figure) and then with RT-DFI (*RT-DFI* in the figure). The mean improvement with both optimizations is 7.6% with a standard deviation of 4.4 percentage points. We note that for 29 benchmarks out of the 47, RT-DFI is the most impactful, while 18 of the benchmarks are more impacted by the value analysis improvements. However, among these 18 benchmarks, 10 have no improvement by RT-DFI because every load on the WCEP has a *valid tag set* with a single tag, thus preventing any further optimization. These 10 benchmarks are *adpcm_dec*, *complex_updates*, *cover*, *deg2rad*, *filterbank*, *fir2dim*, *iir*, *isqrt*, *prime* and *rad2deg*. All these benchmarks have either a very small WCET without protection

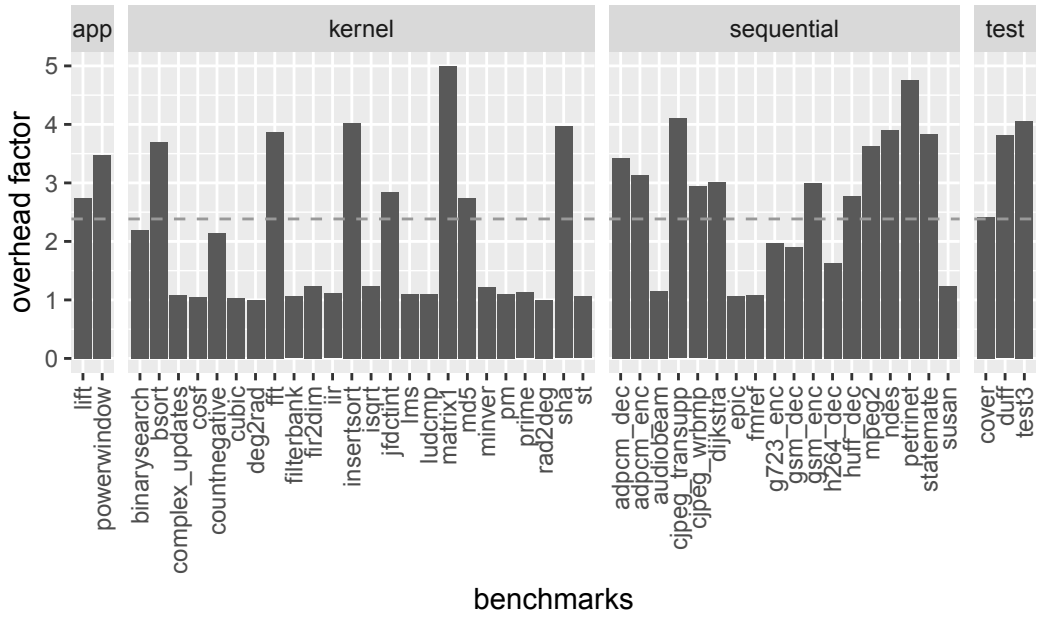

Figure 4 Overhead factor of DFI using the state-of-the-art DFI of [10] with the mean as a gray dashed line.

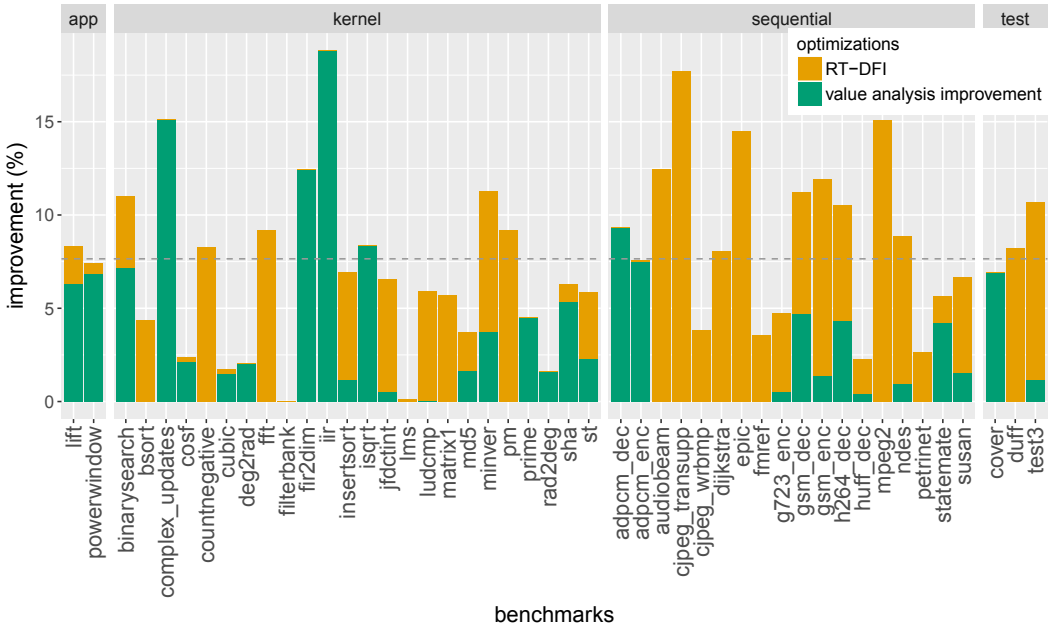

Figure 5 Improvement on the overhead of DFI with value-analysis optimization and RT-DFI. The cumulated mean is represented as a gray dashed line.

or a small DFI overhead (less than $\times 1.24$ if the WCET without protection is higher than $10,000$ cycles). This tends to prove that having very small *valid tag sets* does reduce the overhead to an acceptable level. The overhead of the smallest benchmark skyrockets as they are mostly composed of load/store instructions.

Our optimization process spent most of its runtime executing the WCET analysis ($\sim 66\%$ on average) or compiling the new executable ($\sim 29\%$ on average). For all the benchmarks, the ILP solver part of RT-DFI took less than 40 seconds per problem to solve. Thus, the main runtime cost of RT-DFI is due to our iterative process that requires to re-launch a WCET analysis and re-build an executable at each step. As the number of steps remains low, we do not think this is an issue for the application of RT-DFI. Furthermore, as iterative optimization does not seem efficient, we only need two WCET analysis to perform RT-DFI.

5.3 Notes on iterative optimization

As explained before, we did not include the results for more than one iteration of RT-DFI, as more iterations do not further reduce the WCET. We have two hypotheses as to why iterations do not provide more improvement:

1. The constraints of the previous WCET prevent improvement on the new WCEP.
2. The WCEP has only small changes that have almost no impact on the final WCET[6].

To test hypothesis 1, we relaxed the constraints of the previous WCEPs ($objective_i <= 1.05 \cdot result_i$) to give more freedom to the solver. There was no overhead reduction past the first iteration of RT-DFI. While we can argue if the relaxation is enough and that it should depend on the WCET improvement, this tends to reject hypothesis 1. However, we must remain prudent and more experiments are required to fully reject this hypothesis.

Hypothesis 2 is hard to prove or reject because it is hard to compute path differences between two binaries, where instruction addresses and control-flow graphs may change due to the optimization. We manually examined and saw this problem arise for two benchmarks, namely *lift* and *powerwindow*, by visually examining the Control-Flow Graphs and WCEPs in aiT. However, this method is time-consuming and error-prone. The automatization requires handling changes in the tag representations/interval orders that are equivalent in terms of ILP and/or WCET but may change many parts of the program. As this is a complex issue, we do not have enough data to judge this hypothesis at the moment.

5.4 Notes on the security

Even without mounting a real attack scenario, RT-DFI detected errors on three benchmarks. The first two errors were detected by *aiT* in *rijndael_dec* and *rijndael_enc*. They are due to an off-by-one read to a buffer that loads a stack-saved register and triggers DFI. As *aiT* detects this in its value analysis, it considers that DFI exception is triggered and that its WCET analysis is unsafe. This means that providing *aiT* with a DFI-protected program can help find bugs that can be corrected before reaching the market. The third error appeared when executing *sha* using **Qemu** to test our DFI implementation. DFI detected a read to a saved register when executing the function *sha_wordcopy_fwd_aligned*. This function writes, at each iteration, a word into a buffer and then reads the word for the next iteration. Thus, at the last iteration, a word is read past the buffer, which triggers DFI . Note that

[6] in particular, due to the nature of the benchmarks that contains few paths close to the WCEP

this error is detected at the execution by DFI , but not when using the analysis of *aiT*. This shows that even simple benchmarks can have non-trivial errors that are not detected, and the requirement for improved security protection in RTS.

6 Related work

The security of RTS has gained importance in the last decade. Closely related to DFI, CFI protection has been studied for real-time and embedded systems [1, 24]. Mishra *et al.* have written a survey on CFI techniques for RTS [32]. Using timing information to protect the program has also been studied [4, 41]. With the advancement of attack techniques that bypass *CFI* protection [8, 13], we wanted to focus on stronger protection.

Various implementations and variations of DFI have been studied. Song *et al.* presented HDFI [36], a hardware variation of DFI that uses 1-bit tags to isolate private data. Liu *et al.* studied TMDFI [29], a hardware DFI that modifies a lowRISC core with tag memory to reduce the overhead. Both these approaches modify the hardware and add specific DFI instructions to the ISA. Furthermore, the overhead is hard to predict, which does not fit real-time systems. Feng *et al.* [20] presented a hardware DFI protection that uses processor-in-memory combined with on-the-fly optimizations of memory reads/writes to reduce the memory contention of DFI. As with the previous approaches, specific hardware supports (in particular, processor-in-memory) are required and the overhead is not easily predictable. However, it does not require ISA modification and instead uses standard load/store instructions. Bresch *et al.* presented Trustflow [6], a hardware support for partial DFI with dedicated fast lookup tables. However, it only protects a subset of the data that are selected by hand, it requires hardware support and specific instructions. On the other hand, only protecting a small part of the data remove the overhead. In our work, we have full DFI protection, we do not need hardware support and the overhead is directly obtainable with a WCET solver.

Other methods to protect real-time systems have been proposed. Many researchers have explored vulnerabilities in scheduling methods [11, 27] and mitigations to these vulnerabilities [12, 42, 38, 26, 39, 33, 25]. Fellmuth *et al.* [19] proposed WCET-aware block-level artificial diversity to harden programs in RTS. Burow *et al.* [7] analyzed the impact of current moving target defenses (e.g., ASLR) on the WCET. Kadar *et al.* [24] studied syscall instrumentation to detect attacks in the context of embedded mixed-criticality systems.

7 Conclusion

The security of real-time systems has become an important subject in the last decade. DFI mitigates many potential memory corruption attacks. However, current DFI protections have an overhead that is either very high or hardly predictable. In this paper, we present a method to reduce the overhead of software-based DFI on the WCET. We present an ILP formulation that uses information retrieved from the WCEP to optimize the number of checks of DFI. Our experiment shows that this method helps to reduce the DFI overhead by a mean factor of 7.6%.

Our main limitation is that our method is designed for bare-metal applications. For an independent set of tasks, we could use the same method, with a few tweaks to ensure different tags for each task. However, dealing with tasks sharing resources and *RTOS* still requires more work. In particular, shared resources make the data-flow information used for DFI much harder to obtain precisely.

While our result shows that optimizing the tag checks on the WCET can reduce the overhead by a fair amount, other sources of overhead remain. In the future, we would like to study the overhead due to the imprecision of the data-flow analysis. Another important part of the overhead comes from computing at each store and load the address of the tag. Studying how to reduce this overhead could further reduce the global overhead of DFI. Studying the impact of hardware DFI on WCET also looks promising. Finally, we would like to study the execution of the benchmark applications on a real hardware.

References

1 Fardin Abdi Taghi Abad, Joel van der Woude, Yi Lu, Stanley Bak, Marco Caccamo, Lui Sha, Renato Mancuso, and Sibin Mohan. On-chip control flow integrity check for real time embedded systems. In *1st IEEE International Conference on Cyber-Physical Systems, Networks, and Applications, CPSNA 2013, Taipei, Taiwan, August 19-20, 2013*, pages 26–31. IEEE Computer Society, 2013. doi:10.1109/CPSNA.2013.6614242.

2 Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM. event-place: Alexandria, VA, USA. doi:10.1145/1102120.1102165.

3 anonymous. Morris worm, November 2021. Page Version ID: 1053313243. URL: https://en.wikipedia.org/w/index.php?title=Morris_worm&oldid=1053313243.

4 Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, pages 8:1–8:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ECRTS.2020.8.

5 Christian Bliek1ú, Pierre Bonami, and Andrea Lodi. Solving mixed-integer quadratic programming problems with ibm-cplex: a progress report. In *Proceedings of the twenty-sixth RAMP symposium*, pages 16–17, 2014.

6 Cyril Bresch, David Hély, Stéphanie Chollet, and Ioannis Parissis. TrustFlow: A Trusted Memory Support for Data Flow Integrity. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 308–313, July 2019. ISSN: 2159-3469. doi:10.1109/ISVLSI.2019.00063.

7 Nathan Burow, Ryan Burrow, Roger Khazan, Howard E. Shrobe, and Bryan C. Ward. Moving target defense considerations in real-time safety- and mission-critical systems. In Hamed Okhravi and Cliff Wang, editors, *Proceedings of the 7th ACM Workshop on Moving Target Defense, MTD@CCS 2020, Virtual Event, USA, November 9, 2020*, pages 81–89. ACM, 2020. doi:10.1145/3411496.3421224.

8 Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 161–176. USENIX Association, 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

9 Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

10 Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association. event-place: Seattle, Washington. URL: http://dl.acm.org/citation.cfm?id=1298455.1298470.

11 Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. A novel side-channel in real-time schedulers. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 90–102. IEEE, 2019. doi:10.1109/RTAS.2019.00016.

12 Jiyang Chen, Tomasz Kloda, Ayoosh Bansal, Rohan Tabish, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. Schedguard: Protecting against schedule leaks using linux containers. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 14–26. IEEE, 2021. doi:10.1109/RTAS52030.2021.00010.

13 Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

14 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

15 Irene Díez-Franco and Igor Santos. Data Is Flowing in the Wind: A Review of Data-Flow Integrity Methods to Overcome Non-Control-Data Attacks. In Manuel Graña, José Manuel López-Guede, Oier Etxaniz, Álvaro Herrero, Héctor Quintián, and Emilio Corchado, editors, *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16*, Advances in Intelligent Systems and Computing, pages 536–544, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-47364-2_52.

16 Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, New York, NY, USA, 2015. ACM. event-place: Denver, Colorado, USA. doi:10.1145/2810103.2813646.

17 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

18 N. Falliere, L. O. Murchu, and E. Chien. W32. stuxnet dossier. *Whitepaper, Symantec Corp., Security Response*, 5:6, 2011.

19 Joachim Fellmuth, Paula Herber, Tobias F. Pfeffer, and Sabine Glesner. Securing real-time cyber-physical systems using wcet-aware artificial diversity. In *15th IEEE Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2017, Orlando, FL, USA, November 6-10, 2017*, pages 454–461. IEEE Computer Society, 2017. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2017.88.

20 Lang Feng, Jiayi Huang, Jeff Huang, and Jiang Hu. Toward Taming the Overhead Monster for Data-Flow Integrity. *arXiv:2102.10031 [cs]*, February 2021. arXiv: 2102.10031. arXiv:2102.10031.

21 Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

22 Igor Griva, Stephen G Nash, and Ariela Sofer. *Linear and nonlinear optimization*, volume 108. Siam, 2009.

23 Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. *CoRR*, abs/1711.04808, 2017. arXiv:1711.04808.

24 Marine Kadar, Gerhard Fohler, Don Kuzhiyelil, and Philipp Gorski. Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 292–305. IEEE, 2021. doi:10.1109/RTAS52030.2021.00031.

25 Kristin Krüger, Gerhard Fohler, Marcus Völp, and Paulo Jorge Esteves Veríssimo. Improving security for time-triggered real-time systems with task replication. In *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018, Hakodate, Japan, August 28-31, 2018*, pages 232–233. IEEE Computer Society, 2018. doi:10.1109/RTCSA.2018.00036.

26 Kristin Krüger, Marcus Völp, and Gerhard Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECRTS.2018.22.

27 Jaeheon Kwak and Jinkyu Lee. Covert timing channel design for uniprocessor real-time systems. In Jong Hyuk Park, Hong Shen, Yunsick Sung, and Hui Tian, editors, *Parallel and Distributed Computing, Applications and Technologies, 19th International Conference, PDCAT 2018, Jeju Island, South Korea, August 20-22, 2018, Revised Selected Papers*, volume 931 of *Communications in Computer and Information Science*, pages 159–168. Springer, 2018. doi:10.1007/978-981-13-5907-1_17.

28 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

29 Tong Liu, Gang Shi, Liwei Chen, Fei Zhang, Yaxuan Yang, and Jihu Zhang. TMDFI: Tagged Memory Assisted for Fine-Grained Data-Flow Integrity Towards Embedded Systems Against Software Exploitation. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 545–550, August 2018. ISSN: 2324-9013. doi:10.1109/TrustCom/BigDataSE.2018.00083.

30 Tingting Lu and Junfeng Wang. Data-flow bending: On the effectiveness of data-flow integrity. *Computers & Security*, 84:365–375, July 2019. doi:10.1016/j.cose.2019.04.002.

31 Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

32 Tanmaya Mishra, Thidapat Chantem, and Ryan M. Gerdes. Survey of control-flow integrity techniques for embedded and real-time embedded systems. *CoRR*, abs/2111.11390, 2021. arXiv:2111.11390.

33 Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 103–116. IEEE, 2019. doi:10.1109/RTAS.2019.00017.

34 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for C/C++. In *TACAS (2)*, pages 393–410, 2019.

35 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, October 2004. Association for Computing Machinery. doi:10.1145/1030083.1030124.

36 Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, May 2016. ISSN: 2375-1207. doi:10.1109/SP.2016.9.

37 Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1675–1689, New York, NY, USA, 2017. ACM. event-place: Dallas, Texas, USA. doi:10.1145/3133956.3134026.

38 Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. Avoiding timing channels in fixed-priority schedulers. In Masayuki Abe and Virgil D. Gligor, editors, *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2008, Tokyo, Japan, March 18-20, 2008*, pages 44–55. ACM, 2008. doi:10.1145/1368310.1368320.

39 Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler, and Martina Maggio. Minimizing side-channel attack vulnerability via schedule randomization. In *58th IEEE Conference on Decision and Control, CDC 2019, Nice, France, December 11-13, 2019*, pages 2928–2933. IEEE, 2019. doi:10.1109/CDC40024.2019.9030144.

40 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 Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:24, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.2.

41 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 2012, Karlsruhe, Germany, June 20-22, 2012*, pages 257–266. IEEE, 2012. doi:10.1109/SIES.2012.6356592.

42 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 111–122. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461362.

A Full experimental data

Table 5 Experimental results. **Vanilla** is the WCET of the program without protection. **SotA** is the overhead for state-of-the-art DFI protection. **VA++** is the improvement compared to SotA using the value analysis to improve the valid tag sets. **R1** (resp. **Best**) represents the improvement of RT-DFI with 1 iteration (resp. 4 iterations) compared to SotA.

Bench	Vanilla	SotA	VA++	R1	Best
adpcm_dec	2,306	+243.24%	−9.34%	−9.34%	−9.34%
adpcm_enc	3,306	+213.22%	−7.52%	−7.59%	−7.59%
audiobeam	5,653,355	+15.66%	−0.01%	−12.45%	−12.45%
binarysearch	94	+119.15%	−7.14%	−10.71%	−10.71%
bsort	167,333	+269.44%	0.00%	−4.35%	−4.35%
cjpeg_transupp	10,321,821	+310.02%	−0.00%	−17.72%	−17.72%
cjpeg_wrbmp	165,680	+193.86%	0.00%	−3.83%	−3.83%
complex_updates	23,941	+7.94%	−15.14%	−15.14%	−15.14%
cosf	794,173	+4.77%	−2.10%	−2.36%	−2.36%
countnegative	4,228	+114.50%	0.00%	−8.26%	−8.26%
cover	51	+141.18%	−6.94%	−6.94%	−6.94%
cubic	34,348,748	+3.77%	−1.49%	−1.73%	−1.73%
deg2rad	304,652	+0.03%	−2.04%	−2.04%	−2.04%
dijkstra	3,624,311,302	+200.99%	−0.00%	−8.09%	−8.09%
duff	372	+280.91%	0.00%	−8.23%	−8.23%
epic	11,032,999,977	+7.40%	0.00%	−14.53%	−14.53%
fft	90,043,546	+286.45%	0.00%	−9.18%	−9.18%
filterbank	99,168,970	+5.74%	−0.00%	−0.00%	−0.00%
fir2dim	33,301	+23.43%	−12.42%	−12.42%	−12.42%
fmref	18,678,944	+8.05%	−0.01%	−3.55%	−3.55%
g723_enc	717,091	+96.79%	−0.52%	−4.72%	−4.72%
gsm_dec	1,910,861	+90.82%	−4.71%	−10.90%	−10.90%
gsm_enc	3,719,804	+200.28%	−1.37%	−11.75%	−11.75%
h264_dec	48,385	+62.63%	−4.34%	−10.25%	−10.25%
huff_dec	625,471	+177.46%	−0.42%	−2.28%	−2.28%
iir	5,907	+11.07%	−18.81%	−18.81%	−18.81%
insertsort	1,148	+302.26%	−1.15%	−6.86%	−6.86%
isqrt	698,732	+23.93%	−8.38%	−8.38%	−8.38%
jfdctint	1,593	+184.18%	−0.55%	−6.54%	−6.54%
lift	696,735	+174.71%	−6.33%	−8.22%	−8.22%
lms	3,524,882	+9.65%	0.00%	−0.12%	−0.12%
ludcmp	127,564	+10.44%	−0.03%	−5.92%	−5.92%
matrix1	8,764	+399.36%	0.00%	−5.71%	−5.71%
md5	13,888,069	+174.65%	−1.65%	−3.70%	−3.70%
minver	46,554	+21.97%	−3.74%	−10.97%	−10.97%
mpeg2	2,677,246,923	+263.49%	−0.00%	−15.09%	−15.09%
ndes	116,203	+290.20%	−0.95%	−8.77%	−8.77%
petrinet	2,577	+376.06%	0.00%	−2.66%	−2.66%
pm	197,879,495	+10.17%	−0.00%	−9.18%	−9.18%
powerwindow	2,717,332	+247.30%	−6.86%	−7.37%	−7.37%
prime	1,755	+12.65%	−4.50%	−4.50%	−4.50%
rad2deg	306,691	+0.04%	−1.61%	−1.61%	−1.61%
sha	1,933,236	+296.43%	−5.34%	−6.22%	−6.22%
st	2,991,897	+5.78%	−2.31%	−5.78%	−5.78%
statemate	124,111	+283.22%	−4.21%	−5.56%	−5.56%
susan	388,703,083	+24.24%	−1.55%	−6.57%	−6.57%
test3	235,916,292	+305.47%	−1.17%	−10.57%	−10.57%